Runtime Parallelization of Legacy Code on a Transactional Memory System

Matthew DeVuyst
University of California, San Diego
mdevuyst@cs.ucsd.edu

Dean M. Tullsen
University of California, San Diego
tullsen@cs.ucsd.edu

Seon Wook Kim
Korea University
seon@korea.ac.kr

ABSTRACT

This paper proposes a new runtime parallelization technique, based on a dynamic optimization framework, to automatically parallelize single-threaded legacy programs. It heavily leverages the optimistic concurrency of transactional memory. This work addresses a number of challenges posed by this type of parallelization and quantifies the trade-offs of some of the design decisions, such as how to select good loops for parallelization, how to partition the iteration space among parallel threads, how to handle loop-carried dependencies, and how to transition from serial to parallel execution and back. The simulated implementation of runtime parallelization shows a potential speedup of 1.36 for the NAS benchmarks and a 1.34 speedup for the SPEC 2000 CPU floating point benchmarks when using two cores for parallel execution.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.3.4 [**Programming Languages**]: Processors—*Code Generation, Compilers*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Memory Technologies*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)

General Terms

Design, Performance

Keywords

Parallelization, Dynamic optimization, Transactional memory

1. INTRODUCTION

The multi-core era is upon us, and as feature sizes continue to shrink, dies are divided up into ever more processing cores. The increase in hardware parallelism has outpaced the software industry; there is a large body of applications written for or compiled for a single thread of execution, and these applications do not take advantage of the extra parallelism being offered by modern microprocessors.

The growing need to produce parallel code also motivates us to find better ways of making parallel code easier to write and less error-prone. Transactional memory (TM) is a promising improvement over lock-based synchronization. It optimistically grants access to shared memory, forcing serialization only when a real conflict is detected [5]. TM is an area of active research and has been gaining increasing acceptance in industry [3]. We believe that in the near future we will see more microprocessor vendors including hardware support for transactional memory.

Leveraging this expected hardware support for TM, this paper proposes a new technique to automatically (without any user intervention) extract thread-level parallelism from legacy single-threaded programs with minimal architectural change. We find that transactional memory enables the parallel execution of many loops that are serialized by traditional synchronization. Our technique uses a dynamic optimization framework: frequently executed loops are identified by hardware at runtime, a dynamic re-compiler is spawned in a free hardware context to transform the key loops into parallel code, and the re-compiled parallel loops are patched in to the running program. The dynamic re-compiler analyzes the loop (at the machine code level, not at the source code level) and, if possible, transforms it into a loop that can be executed in parallel. When the parallelized loop executes, parallel threads are forked onto free hardware contexts.

The primary contribution of this paper is to show that the reduced overhead and optimistic concurrency of hardware transactional memory enables the effective parallelism of a number of legacy codes, despite the existence of unknown and unknowable memory aliasing.

This paper describes our parallelization technique and quantifies its effectiveness across a number of benchmarks. It is organized as follows. Section 2 is an overview of the related work. Section 3 describes our baseline processor architecture, including transactional memory and dynamic optimization implementations. Our parallelization technique is described in Section 4. An important part of our parallelization, code generation, is described in Section 5. Section 6 describes our experimental methodology. In Section 7 we present our results. Section 8 concludes.

HiPEAC 2011 Heraklion, Crete, Greece

2. RELATED WORK

2.1 Transactional Memory

Our parallelization technique relies on detection and correct execution of memory-dependent computation. This requires a mechanism to prevent erroneous execution that arises from unanticipated memory aliasing between loop iterations. We achieve this through the use of transactional memory.

Transactional Memory has been proposed as a hardware implementation of lock-free concurrent synchronization [5]. The TM model provides the programmer with guarantees about the memory accesses of instructions contained in transactions. One guarantee is that writes to memory within a transaction are not visible to other transactions until transaction commit (at which time all the writes in the transaction are atomically released). Another guarantee is that memory aliasing among transactions (e.g., a write in one transaction to the same address as a read in a concurrently-executing transaction) are detected and appropriate recovery actions taken (e.g., a transaction's state will be discarded and it will be restarted).

Both hardware [5] and software [13] transactional memory implementations have been proposed. We assume hardware transactional memory, but otherwise, we are relatively insensitive to which implementation. We only require that the TM implementation support ordered transactions. Ordered transactions are only allowed to commit in a pre-defined order; they have been proposed in [4, 1, 11].

2.2 Dynamic loop parallelization

Other researchers have proposed the use of dynamic optimization to aid in runtime parallelization of loops.

Ootsu, *et al.* [9] propose a runtime parallelization technique for binaries using a dynamic optimization framework. Their work builds on the parallelization technique of Tsai, *et al.* [16] by adding binary translation at runtime instead of transformation at compile-time to perform the parallelization. Unlike our work, they don't explore the use of any optimistic concurrency mechanisms to ensure correctness with regard to concurrent memory access.

Yardimci and Franz [20] put forth a dynamic parallelization and vectorization technique for single-threaded binaries. Their technique involves only control speculation, not data speculation. To this end, they introduce a complex control flow analysis (especially to handle indirect branches); but they do not parallelize loops whose induction registers do not have deterministic fixed strides, loops that have stores in conditionally-executed code regions, or loops with potential cross-iteration data dependencies. Because our approach leverages transactional memory for data speculation, we find ways around all these limitations and are thus able to parallelize more loops.

Vachharajani, *et al.* [18] add speculation to Decoupled Software Pipelining (DSWP). DSWP parallelizes a loop by partitioning the loop body into stages that are scheduled on threads and executed in a pipelined manner, communicating results via a message passing or buffering mechanism. Speculative DSWP aims to break some recurrence dependencies so that the loop body can be broken into smaller pieces to increase scalability and load balancing. Our work differs from theirs in three ways: our technique works at the machine code level, where analysis and transformation is more complicated due to compiler optimization and loss of information; we use a simpler, more common form of speculation, Transactional Memory—they use a complicated versioned memory system; their technique utilizes heavy inter-thread (cross-core) communication, potentially requiring hardware support for best performance.

```
loop: sub r1, r1, 256
      add r5, r1, r6
      load r2, 64(r5)
      load r3, 0(r2)
      add r3, r4, r3
      store r3 0(r2)
      bne r1, 0, loop
```

Figure 1: Potential memory aliasing makes this loop hard to parallelize.

Works by Zhong, *et al.* [23] and Von Praun, *et al.* [19] leverage TM for parallelization, but rely on programmer and/or heavy compiler support; our technique requires neither.

Speculative multithreading [15, 7, 6] is an alternate approach that attempts to get parallel speedup from serial code. They do so by executing serial code in parallel and recover from misspeculation. They typical rely heavily on data value speculation and prediction, as well as some type of memory versioning. Conversely, we create more conventional parallel code (dynamically), without data prediction, and only exploit speculative execution to the extent that transactional memory already supports it.

3. ARCHITECTURE

Hardware transactional memory provides several key advantages over traditional synchronization, particularly with regards to the problem of runtime parallelization. The key problem in runtime parallelization that has made it an essentially unsolvable problem except in the most simple cases, is that in the absence of any high-level program information, nearly all loads and stores must be treated as potentially aliased—the necessarily conservative handling of these potential dependences serializes the code. Transactional memory, by supporting optimistic concurrency, solves a whole set of problems. First, because code is only serialized when there is true aliasing, conservative placement of synchronization has no cost. Second, we can include many writes in a single transaction, minimizing synchronization overhead with no significant loss in concurrency. Because transactional semantics requires no correlation between the synchronization mechanism and the data that is protected, parallelization is simply enabled yet still catches even unanticipated dependences.

To illustrate the power of this technique, consider the pseudo-assembly code of the loop in Figure 1. This code loops over an array of structures: for each structure element, a pointer field is extracted and followed, and the data at that pointer location is modified. Assume this loop is executed many times, is part of a single-threaded application (for which we do not have the source code), and it is running on a modern multi-core processor on which there are one or more unutilized cores.

Using hardware performance counters, a dynamic optimization framework can detect that this loop executes fre-

```
        fork fix
        branch btx
loop:   EndTransaction
fix:    sub r1, r1, 256
btx:    BeginTransaction
        sub r1, r1, 256
        add r5, r1, r6
        load r2, 64(r5)
        load r3, 0(r2)
        add r3, r4, r3
        store r3 0(r2)
        bne r1, 0, loop
        EndAllTransactions
join:   ...
```

Figure 2: Wrapping the loop body in a transaction allows for safe and optimistic parallelization.

quently and can transform it automatically into parallel code—with different iterations of the loop running in different threads. Without the high-level code, we cannot guarantee that parallel iterations of this loop won't attempt to modify the same data in memory. With traditional lock-based synchronization primitives, our parallelizer would transform the loop such that a lock would have to be acquired and released on every iteration of the loop. Since only one thread may hold the lock at any time, even though the loop would be parallelized across multiple threads, the frequent synchronization would force a serialization of loop iterations. However, if the loads and stores are not frequently aliased in neighboring iterations, the serialization is unnecessary.

If the optimizer instead uses transactional memory to make the code thread-safe, then when aliasing is infrequent, unnecessary serialization does not hinder performance. Each iteration is wrapped in a transaction and the transactions execute concurrently. Thus, with minimal analysis of the code, we still get guaranteed serialization of iterations when there are dependences, and parallel execution in the absence of dependences.

3.1 Code Transformation Overview

We now present a high-level overview of the code transformation. Section 4 provides more details on this process.

Figure 2 shows what the transformed code from Figure 1 would look like, when targeting two parallel threads. Some details (mostly for bookkeeping, such as numbering transactions for ordering) have been omitted to simplify the example. The first thing this code does is fork a new thread to start executing at the label *fix*. This is a lightweight fork—no new stack is created, only registers are copied and the PC is set. The newly-created thread will execute the induction code to bring the loop-carried registers up to date in preparation to execute the second iteration of the loop: in this example, 256 is subtracted from register r1. Then a new transaction is begun and the loop body is executed. Meanwhile, the original thread will branch to the label *btx*, open a new transaction and execute the first iteration of the loop. When each thread completes an iteration, i, of the loop, it will close the transaction and open a new transaction to execute iteration $i + 2$. The ordering of the transactions will ensure that state from each iteration is committed (i.e., data is stored to memory) in original program order. When the loop is done executing, all the transactions will be closed, the spawned threads will be terminated, and execution will resume in the original thread at the *join* label.

Having ordered transaction commits ensures that we do not commit values before all prior execution commits, and also ensures that the last write of each memory location in the loop is the write that is visible at loop exit.

4. PARALLELIZATION

There are several things we need to enable dynamic parallelization of legacy serial code. First, we need some kind of dynamic optimization framework that can (1) identify candidate code to parallelize, (2) spawn a thread to analyze the code and generate parallel code, and (3) patch in the new code. We also need hardware support to catch dependences between iterations assumed to be parallel (transactional memory, in our case). We need our parallelizer to solve the problem of selecting the right granularity of parallelism. We also need to ensure that we maintain the semantics of sequential execution, particularly as viewed by the code following the parallel region—transactions accomplish this for memory, but registers must be handled in software. Our solutions for each of these issues will be presented in this section and the following one. This section focuses on the overall design of the parallelization process, including the dynamic optimization framework that orchestrates the parallelization process and the scheduling of parallel code on threads. The next section will focus on aspects of the parallel code generation itself.

4.1 Dynamic Optimization Framework

The basic unit of optimization targeted by our technique is loops. To identify loops most effectively, we combine two techniques—a whole-program control-flow analysis that identifies loops, combined with a hardware monitor that identifies frequent branches. This provides a more accurate view of important loops than trying to identify the loop based on a hot-branch address.

When a new process is started, a loop analyzer begins in a spare hardware context to perform a quick loop analysis of the binary code. Since this is a static analysis, it can be performed once and the results even saved to a file for all future executions. The loop analyzer builds a dominator graph among basic blocks to find natural loops [8]. It excludes loops that contain system calls or computed branches (because parallelizing such loops would be problematic). The static analysis is not required, but improves the quality of discovered loops. Because it runs in a separate context and typically requires only a few milliseconds, it neither slows the main thread nor impedes our dynamic parallelizer in any but the shortest of applications.

As the program executes, a hardware-based profiler, like the Hot Path Profiler in [21], finds frequently executed loops, called *hot* loops. When a hot loop is identified, hardware monitors measure the average number of cycles required to execute one iteration of the loop. Once the baseline performance of the hot loop has been measured, a dynamic re-compiler is spawned in a spare hardware context to attempt to parallelize the hot loop. The operation of the dynamic re-compiler will be described in the next section.

One of the first few instructions in the new version of the loop is a fast fork instruction. When this instruction is executed, a new thread is created on a spare hardware context. This is not a traditional heavy-weight fork. No

new stack is created for the new thread and no registers are set, except for the program counter, which is set by adding the current PC to the offset encoded in the fork instruction. Because only the PC has to be set in the newly-created thread, the fork can be very fast.

As the parallel version of the loop executes, its performance is monitored in hardware. If the parallel version of the loop is not performing any better than the serial version, the parallel version is eventually removed, as in [21]. When all iterations of a dynamic instance of a parallelized loop complete, all forked threads are terminated, and serial execution continues on the original thread.

4.2 Partitioning the iteration space

Loop iterations are distributed among threads in a round-robin fashion. In our baseline implementation, this distribution is done at the granularity of individual iterations. We also experiment with distributing groups of iterations among threads (called *tiling*), where each group is wrapped in a single transaction. The *tile size* is the number of consecutive loop iterations grouped together and treated as a unit of parallel work. Larger tile sizes can reduce transaction overhead and increase cache locality.

Each loop has an ideal tile size based on factors such as iteration count, loop body size, induction code size, transactional overhead cost, probability of early exit, and probability of transaction restart. For most loops, tile size does not significantly affect performance—other factors, like loop-carried dependencies and transaction restarts play a much greater role in determining the performance of a parallelized loop.

Given the marginal performance improvements of finding optimal tile sizes and the significant increase in the complexity of analysis required (which is contrary to the design philosophy of our *fast* dynamic re-compiler), we do tile size selection two ways. First, we can statically select a tile size that has been found to result in good performance, in general, across all applications. For the results presented in this paper, the tile size we selected was 16. Alternatively, we can use the dynamic optimization framework to iteratively re-parallelize loops and sample performance to discover better tile sizes.

5. CODE GENERATION

This section describes the parallel code generation. There are a number of necessary features to parallel code generation. Parallel code must be able to take advantage of cross-iteration dependency checking (enabled by transactional memory). It must maintain the semantics of sequential execution, handling explicitly what TM does not—data flow through registers. And it must have some way of dealing with function calls in parallelized code.

5.1 Transaction wrapping

The use of transactional memory in automatic parallelization is a key element of our study. The key benefit provided by TM is optimistic concurrency among parallel threads in the face of statically unknown memory sharing. While loop-carried dependencies though registers can be determined statically, dependencies through memory cannot always be determined a priori. Some memory sharing can be determined statically, but this can require a complicated and costly analysis. The analysis is further complicated by the applications we are targeting—single-threaded legacy binaries with no source code available. For all these reasons, our dynamic re-compiler analyzes loop-carried dependencies through registers but does not explicitly analyze memory sharing among parallel threads, relying on the TM system at runtime to detect and recover from memory violations caused by inter-iteration memory sharing.

Every loop iteration (or consecutive group of loop iterations if tiling is used—see Section 4.2) is wrapped in an ordered transaction. The ordering of the transactions match the original sequential ordering of the loop iterations.

5.1.1 Opening transactions

Individual transactions are started with the BTX (Begin Transaction) instruction; this signals the Transactional Memory Manager (TMM) that a new transaction is starting. In our TM model, as in most TM proposals, register state is saved when a transaction is begun in case the transaction fails to commit and has to be rolled back. We assume a shadow register file for this purpose. Memory instructions executed after the beginning of a transaction are said to execute in the transaction and follow the transactionally-extended cache coherence protocol (transactional bits are marked as appropriate and written memory is prevented from leaving the local cache).

5.1.2 Closing transactions

Transactions can end in three ways. The most common way a transaction ends is by committing. When a transaction commits, the memory state written inside the transaction is made available to other threads. In the case of our cache-based TM implementation this means that cache words marked transactionally written can be copied to private caches of other cores when requested, or written back to the L2 cache if evicted. Most transactions commit when they reach an ETX (End transaction) instruction and all older transactions have committed. If an ETX instruction is reached and there are uncommitted older transactions, the thread executing the transaction is stalled. An ETX instruction is executed at the end of every loop iteration (after the branch to a new iteration has been taken).

The second way that a transaction may end is by an explicit abort. We've implemented this functionally in the ATX (Abort Transaction) instruction. This instructs the TMM to invalidate all the local cache words marked as transactionally written, effectively rolling back the memory state of the transaction (and the register state via the shadow register file). Explicit aborts also obey the transaction ordering semantics—no transaction may abort unless all older transactions have committed. Otherwise, speculative transactions with invalid data could cause aborts. ATX instructions are inserted down loop early-exit paths. An early exit path is the path taken when a loop exits at a point different from the loop continuation. A C/C++ *break* instruction in the middle of a *for* loop is one example of an early exit.

The third way that a transaction may end is by a transaction restart. Restarts are triggered when a transactional memory violation is discovered (for example, if it is discovered that a store instruction in an older transaction wrote to an address that a load in a younger transaction read from). Violation detection is built into the cache coherence protocol and happens eagerly (at execution of the violation-causing memory instructions) instead of lazily (at the end of the

transaction). This means that a transaction may restart at any time during execution. The semantics of a restart are virtually identical to those of an explicit abort: cache words marked transactionally written are invalidated and register state is restored. Execution then starts again from the BTX instruction that began the transaction.

5.2 Data flow through registers

Dependencies and data sharing through memory among iterations are handled by the transactional memory system at runtime, but dependencies and data sharing through registers must be analyzed by the dynamic re-compiler and handled explicitly in the parallelized code. Dependency analysis involving only registers is significantly less complex than memory analysis, but there are several challenges to generating parallel code out of sequential. Loop-carried dependences through registers are problematic because cores do not share registers. Also, on loop exit, the continuing thread must see a single register file with the latest update to each register, even though the last write could have occurred in any core. The latter issue requires register value updates when (1) register values are conditionally written, or (2) when the last iteration of the loop does not execute on the original thread.

5.2.1 Loop-carried register dependencies

We can manage most loop-carried dependencies by treating the dependent computation as induction code. When the dynamic re-compiler analyzes a loop before parallelization, in addition to building a control-flow graph for the loop, it builds a data-flow graph to model the flow of data though registers. A list of registers that are loop-carried is generated by identifying all registers that are read before being written to in the loop body. Induction code is extracted from the loop by following the data-flow graph backwards, starting at the last writes to each loop-carried register, including every instruction that is necessary to compute the new values of the loop-carried registers. This induction code is added to the beginning of the loop body to generate the correct live-ins.

5.2.2 Conditionally-written registers

Registers written in conditionally-executed basic blocks are problematic for two reasons. When they are loop-carried registers, the conditional statements can cause significant induction code expansion. For non-loop-carried registers, correctly identifying the last write to the register is difficult. In both cases, we exploit the transactional memory system's facility of guaranteeing correct ordering.

To keep the induction code small, we have chosen not to include control flow in induction code. Instead, both loop-carried registers and non-loop-carried conditionally-written registers are passed through memory and no induction code needs to be generated for them—transactional memory will ensure correct ordering. This is enabled though the use of spill (store) and fill (load) instructions. Registers that are last written in a conditional block are spilled to memory. If the register is loop-carried as well, a fill instruction for that register is inserted at the beginning of the loop. For all conditionally-written registers, a spill instruction is inserted in the loop prologue (to be executed before any iterations) and a fill instruction is inserted in the loop epilogue and down early exit paths. For loop-carried, conditional register dependences that are infrequent, this allows the code to typically execute in parallel. When the dependences are frequent, there will be frequent restarts and hardware monitors will recognize our failure to achieve speedup on the loop; this will cause the loop to revert to the original code.

```
while (i < 1000) {
  old_count = count;
  if (A[i] == '!') {
    count++;
    last = i;
  }
  ...
}
```

Figure 3: C code to illustrate two kinds of register writes. Assuming that the compiler binds *count* and *last* to registers, the registers representing these variables are conditionally-written. *count* is loop-carried and *last* is not.

The C code in Figure 3 illustrates both kinds of conditionally-written registers. This loop iterates over a character array and counts the number of exclamation marks found, recording the location of the last mark for later reference. Assuming the variables *count* and *last* are bound to registers, *last* represents a non-loop-carried conditionally-written register and *count* represents a loop-carried conditionally-written register. The dynamic re-compiler will insert two register spill instructions at the end of the basic block where *count* and *last* are conditionally written to store the value of *count* and *last* to memory. If the last iteration where *last* is updated is executed on a forked thread, that thread can be safely terminated and the correct value of *last* filled from memory by the original thread after loop termination. The value of *count* is filled from memory at the beginning of every iteration. In the common case, where the last character was not an exclamation point and the conditional code was not executed, the private caches of all parallel threads will read-share this value and the same value will be read every time. When the conditional code is executed and the new value of *count* is spilled to memory, parallel iterations that logically follow that iteration will be restarted and will then read the correct value for *count*.

Another type of control flow that complicates register dependence handling is nested loops. If a loop-carried register dependence is in a nested loop or depends on code in a nested loop, then induction code would have to include the nested loop; however, this is undesirable. Using the spill/fill technique that we use for conditional code would hurt performance as well. As a result, we do not currently parallelize loops when the outer loop induction code would include register values computed in an inner loop.

5.2.3 Register state of the last iteration

We must also identify the last writer of registers that are not conditionally written. This is an easier problem, but may still involve data transfer. Unlike conditionally-written registers, at loop termination we know exactly which thread had the correct, most up-to-date value of the non-conditionally-written registers: the thread that executed the last iteration of the loop. This allows us to track register values without any restart-inducing data sharing. At the end of every loop iteration, register state is spilled to a thread-

specific memory location; no cache lines have to be write-shared by different threads for this. Upon loop termination, after the forked parallel threads have been halted, the original thread fills the register state from the memory location of the thread that executed the last loop iteration. The dynamic re-compiler inserts the fill code in both the loop epilogue and the early exit code.

5.3 Function calls

Function calls in loops introduce a difficult challenge for a number of reasons. For example, we use a lightweight fork mechanism, as described, that does not require us to allocate a new stack for the forked threads—they all share the same stack space. Because function calls grow the stack and allocate stack-based variables, allowing function calls in parallel loop iterations is problematic. Additionally, we cannot allow our parallel code to call an uninstrumented function unless we can guarantee that it returns to the call site. When dealing with machine code, this is harder to guarantee than with high-level source code; this is particularly true when the return address is written to memory (because it is difficult to guarantee that that memory is not modified).

Both of these problems can be solved by inlining functions. An inlined function is not allocated its own stack frame—so we don't have to deal with parallel threads allocating new stack frames. The second problem is solved because as the dynamic re-compiler analyzes and prepares a function for inlining, it ensures that it returns to the call site.

Our re-compiler inlines one level of function calling. Any function calls within called functions are replaced with early exit points that facilitate the transition from parallel execution back to serial execution at the point of the function call site. If the second-level call is only conditionally executed, the early exit may not inhibit parallelism severely.

Also, function inlining allows us to easily modify a copy of the function intended for parallel execution without making changes to the original copy of the function intended for sequential execution (when accessed by other call sites).

6. EXPERIMENTAL METHODOLOGY

This initial study of the use of hardware transactional memory to facilitate automatic runtime parallelization of legacy code is in some respects a feasibility study to determine to what extent we can expose the available parallelism. For that reason, we keep our execution model relatively simple—we assume two cores. One core runs all the serial portions of code as well as one of the parallel threads in parallelized code regions. The other core executes parallel threads when available and the dynamic compiler thread (which only runs about 3% of the time). The maximum expected parallel speedup is 2.0.

Each core is an in-order core. See Table 1 for more details of our processor architecture. In the following section, we will describe the architecture of our transactional memory model.

6.1 Transactional Memory

We model a generic transactional memory system. The only relatively uncommon characteristic that we require is the support for ordered transactions. Many proposed transactional memory systems view all concurrent transactions as equal; however, to preserve program order we require preferential treatment for transactions executing earlier (less-speculative) loop iterations. The ordering of transactions is considered in decisions regarding which transaction(s) to restart when a memory violation occurs, and in restricting when a transaction can commit. The modifications necessary to add ordering among transactions are straightforward and a number of transactional memory proposals support ordered transactions [4, 1, 11].

Cores	2	Shared L3 cache	4M, 2 way
Total Fetch Width	4	L1-L1 transfer	14 cyc
Int/FP regs/core	100/100	Load-use, L1 hit	2 cyc
I cache/core	64k, 2 way	Load-use, L2 hit	16 cyc
D cache/core	64k, 2 way	Load-use, L3 hit	58 cyc
Shared L2 cache	512k, 2 way	Load-use, L3 miss	158 cyc

Table 1: Architecture Detail

We have found that memory violation detection at cache line granularity results in poor performance for many loops due to frequent transaction restarts caused by false sharing. Consider the case of a very simple loop that writes successive elements in an array of integers. If each iteration is wrapped in a transaction and executed concurrently, conflict detection at cache line granularity would result in false sharing as each parallel thread tries to write to different parts of the same cache line. Because of the significant performance advantages offered by violation detection at word granularity, we will assume this granularity. Note that our parallelization technique is compatible with cache line granularity violation detection and will still yield a speedup, just not as great. A further discussion of some of the trade-offs of granularity and a performance comparison can be found in Section 7.2.

In trying to model as generic a Transaction Memory system as possible, we assume a TM system that is similar to the simple original model proposed by Herlihy and Moss [5]. Like many newer TM proposals, we model the buffering of transactional state in the local caches instead of a special Transactional Buffer. The traditional MESI cache coherence protocol [10] is extended to support memory violation detection (as is done in [4, 11], for example).

6.2 Simulation and Benchmarks

For these measurements, we simulate steady-state execution well into the program. Thus, we assume that prior to the measurement interval, the one-time static program analysis has completed, and that loops that our system would try to parallelize but fail to achieve speedup have already been identified and rejected. The mechanics and efficiency of the code-cache uninstallation process has been shown in prior work (e.g. [21]). Thus, our results are somewhat optimistic; but in that prior work it was found that after an initial warm-up, the code changed very infrequently.

We implemented the dynamic re-compiler to parallelize loops in binaries compiled for the Alpha architecture. The dynamic re-compiler was not designed to be a robust full-featured compiler, but to be very small and lightweight, allowing it to quickly re-compile loops at runtime.

We extended SMTSIM [17], an event-driven Chip Multiprocessor (CMP) and Simultaneous Multithreading (SMT) processor simulator, to support transactional memory and a dynamic optimization framework (a simplified version of the one by Zhang, *et al.* [21]). The simulator was configured as a CMP and executes Alpha binaries, including our dynamic

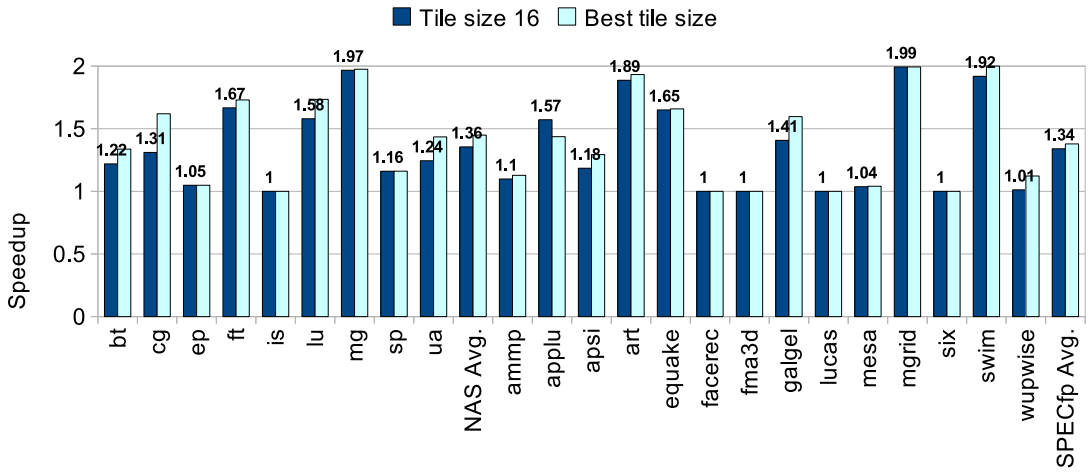

Figure 4: Speedups of NAS and SPEC FP benchmarks. The dark bars and data labels show performance when the tile size is fixed at 16. The light bars show performance when an optimal tile size is selected for each loop.

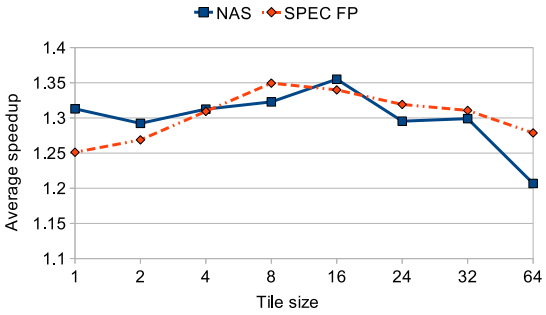

Figure 5: Speedup across various tile sizes.

re-compiler compiled to the Alpha ISA.

Our primary target is legacy code, and in particular we want to address two types of applications—those for which thread-level parallelism is clearly available but the code was compiled single-threaded to run on legacy processors, and those clearly written for single-threaded execution, but some thread-level parallelism may still be available. We use NAS 3.3 benchmarks to represent the former (except *dc* which did not compile properly for Alpha OSF/4 using GCC) and all the SPEC2000 CPU floating-point benchmarks to represent the latter. All the benchmarks were compiled with GCC 4.3 at optimization level -O2. Even when thread-level parallelism was clearly evident in the source code, we found that compiling for a single thread at this high optimization level seriously obfuscated that parallelism in many cases.

The NAS benchmarks were each fast-forwarded one billion dynamic instructions before detailed simulation in order to skip over program initialization. A SimPoint ([14]) analysis was performed for each SPEC benchmark to find representative points of execution for detailed simulation. The A inputs were used for the NAS benchmarks and the reference inputs were used for the SPEC benchmarks. For the SPEC benchmarks that have multiple reference inputs, the first (alphabetically ordered) inputs were used.

For the results we present in this paper, loops were parallelized into two threads and the default tile size was 16.

7. RESULTS

The performance gains of our parallelization with constant tile size are presented in Figure 4 (the dark bars). The average speedup among the NAS benchmarks is 1.36 and the average speedup among the SPEC FP benchmarks is 1.34. The performance of some benchmarks, like mg, mgrid, and swim came close to the theoretical limit of 2X speedup. Others, like facerec and fma3d (to name a couple), saw no performance gain. In general, despite the challenges of identifying, transforming, and exploiting parallelism in serial code at runtime, we were successful, to some degree, in a significant percentage of the applications.

There are three primary reasons why some benchmarks could not benefit from parallelization. First, in some cases, there is an inherent lack of thread-level parallelism. In other cases, thread-level parallelism is present, but it is not expressed in a way that is amenable to our parallelization technique. For example, a reduction [12], like a loop that sums up values in an integer array, has inherent parallelism. But if it is not coded carefully to express that parallelism, a critical loop-carried dependence will be created between every consecutive iteration of the summation. Since our dynamic re-compiler operates at a very low level, it is not always able to recognize and transform every expression of parallelism. Third, and probably most importantly, some optimizations (like software pipelining) and machine code generations during original compilation obscure or impede subsequent parallelization. Thread-level parallelism that is evident in the high-level language of the source code may not be obvious or may not exist in the optimized single-threaded binary. The poor performance on *ep* is clearly an example of this case.

7.1 Tile size

Figure 5 demonstrates how variance in tile size affects performance. For benchmarks in both suites, some degree of tiling is necessary in order to obtain the highest performance gains; however, we find that tiling is not absolutely necessary to see reasonable performance gain. The best tile

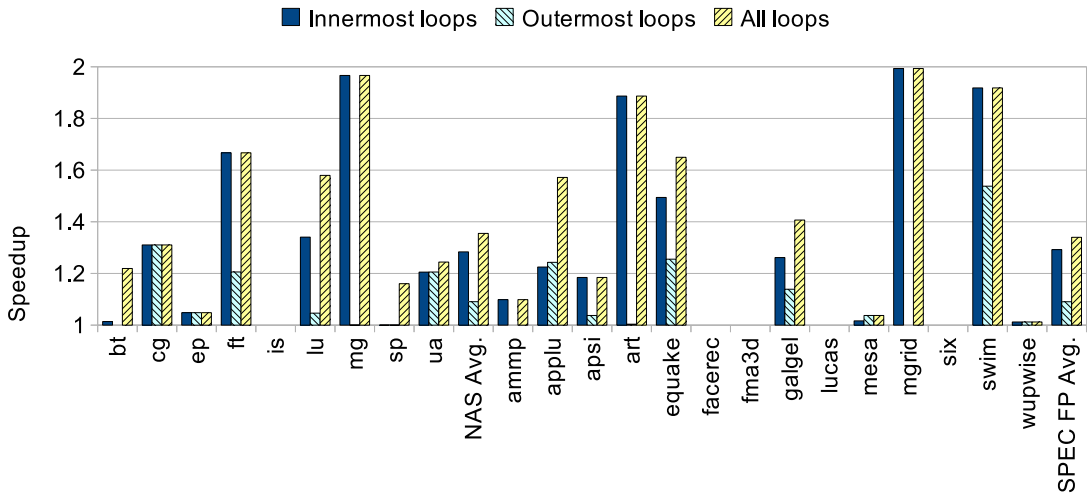

Figure 6: Speedups of NAS and SPEC FP benchmarks under different loop selection policies: innermost loops only, outermost loops only, and unconstrained.

size for NAS benchmarks was 16, while the best tile size for SPEC FP benchmarks was 8. As tile size increases much beyond 16, performance decreases in spite of the decreasing parallelization overhead. Because transactions are larger, transaction restarts are more costly because more worthless instructions are executed—instructions whose effects will be undone. Also, high tile size often creates a load imbalance among parallel threads during the last iterations of the loop.

When the tile size of each loop was varied and the best-performing parallel loop version was selected, the distribution of best tile sizes was highly distributed: 19% of the time the best tile size was 1, 9% of the time it was 2, 9% of the time it was 4, 15% of the time it was 8, 8% of the time it was 16, 14% of the time it was 24, 11% of the time it was 32, and 15% of the time it was 64. These results imply that for best performance, the dynamic optimization framework should attempt parallelizations at a variety of tile sizes. Conversely, if loop analysis can find the optimal tile size, it could reduce the number of parallelized loop versions to be tried. Event-driven dynamic compilation has been shown to be quite effective at just this kind of trial-and-error optimization [22], due to the very low overhead of recompiling threads that run in another core.

The light bars in Figure 4 show speedup when the optimal tile size per loop is selected. The average NAS benchmark speedup is 1.45 (a 9% performance improvement) and the average SPEC FP speedup is 1.38 (a 4% performance improvement). The performances of most benchmarks improve very little when optimal tile sizes are selected on a per-loop basis instead of a single static tile size. However, a few benchmarks, like cg and galgel, showed a more significant performance improvement. The performance of applu is slightly lower due to a negative interaction between some of the loops of different tile sizes (minor second-order caching effects) that our loop selection mechanism wasn't able to detect a priori.

In all of the results we have presented so far, parallel loops of various degrees of nesting were installed. We experimented with two (more constrained) selection policies. In the first of these policies the dynamic optimization framework selected only parallelizable innermost loops. Loops that had no nested inner loops were included in this selection. In the second policy, only outermost loops were selected. Loops that were not nested within any other loops were included in this selection. Based on how we have defined these policies, loops that are not nested in any other loops and do not contain any nested loops themselves are selected in both policies. Note that there are some loops that fall in neither category: loops that are contained within another loop and have a nested inner loop. We compared the performance of parallelization under these two policies and our unconstrained policy.

The performance results are given in Figure 6. For most benchmarks, parallelization of the innermost loops results in better performance than parallelization of the outermost loops. There are three reasons why innermost loops are generally better candidates for parallelization: the thread level parallelism our technique exploits most readily is fine-grained; fine-grain thread-level parallelism is more abundant than coarse-grain thread-level parallelism in the code we run; also, because the size of outer loops are greater, the restart cost of a failed transaction is greater. Larger loops have a greater potential for inter-iteration dependencies, resulting in more transaction restarts that hurt parallel performance. This figure, then, shows two key results. If we are trying to really minimize the complexity of the dynamic compiler threads, restricting it to innermost loops is very effective. However, there is a reasonable gain to considering all loops as candidates.

7.2 TM cache granularity

All of the results shown so far have been with the transactional memory system configured at word granularity instead of cache line granularity. Word granularity will be more expensive, but will yield performance results even on traditional transactional code [2]. We explore how that granularity affects performance.

Figure 7 compares performance across all the benchmarks when transactional memory is implemented at word granularity verses cache line granularity. The average speedup

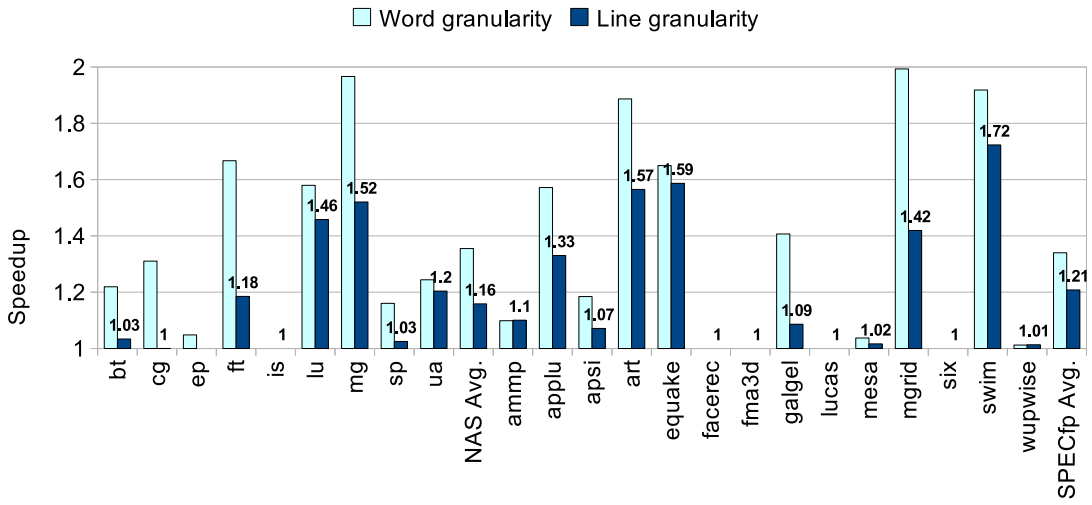

Figure 7: Speedups of NAS and SPEC FP benchmarks. The light bars show performance when TM is implemented at word granularity. The dark bars and the data labels show performance when TM is implemented at cache line granularity. The tile size in both cases is 16.

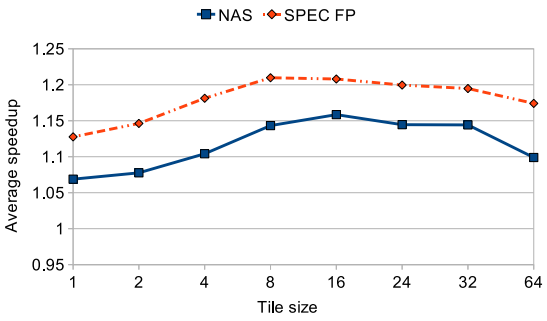

Figure 8: Speedup across various tile sizes when TM is implemented at cache line granularity.

across the NAS benchmarks drops from 1.36 with word granularity to 1.16 for line granularity; and the average speedup across the SPEC floating-point benchmarks drops from 1.34 to 1.21. While TM at word granularity increases data cache complexity and size, it dramatically increases parallel performance. TM at cache line granularity causes significant false sharing, both write-after-write (WAW) false sharing and read-after-write (RAW) false sharing. This increases the transaction restart frequency which reduces parallel performance.

When TM is implemented at cache line granularity, small tile size results in more frequent false sharing. At higher tile sizes, some false sharing can be eliminated as parallel threads are more likely to operate on data in disjoint cache lines. This affect is highlighted in Figure 8. Here we see that performance is worse at lower tile sizes. We also note that across all tile sizes, the average performance of the NAS benchmarks is distinctly less than the average performance of the SPEC FP benchmarks. This is not something we observe at word granularity. This suggest that the NAS benchmarks are more prone to false sharing.

Despite the expected frequency of false sharing between iterations of legacy code compiled for single-threaded execution, we still do see significant (albeit reduced) opportunity to find and exploit parallelism. This is because we assume a system that can identify poor-performing loops; thus, loops with frequent false-sharing induced transactional restarts will be discarded.

8. CONCLUSION

In this paper we present a runtime parallelization technique that leverages the expected upcoming support for transactional memory and the runtime flexibility and efficiency of dynamic optimization. It allows single-threaded legacy binaries to achieve performance improvements in the increasingly common context of multi-core microarchitectures. Our parallelization technique makes use of transactional memory to provide optimistic concurrency and to make strong guarantees about correctness in code that a traditional compiler would have a hard time proving correct. Furthermore, parallelization is accomplished without any need for assistance from the user or programmer and without access to the original source code.

We discuss some of the unique challenges posed by runtime parallelization and show how we address these challenges in our design. Our runtime parallelization shows the potential for 36% performance improvement across the NAS benchmarks and 34% performance improvement across the SPEC2000 floating-point benchmarks, utilizing two-core parallelism. We show that a loop selection policy that considers only loops at a particular nesting level (e.g. innermost loops only) fails to achieve the highest performance. We show that most applications are fairly intolerant of tile size (the number of iterations per transaction). They are more sensitive to the granularity of the underlying transactional memory system, achieving significant gains when conflicts are detected at a word granularity.

9. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for many useful suggestions. They would also like to thank Leo Porter for his help with the Transactional Memory model. This work was supported in part by NSF grant CCF-0702349 and a gift from Intel.

10. REFERENCES

[1] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, 2006.

[2] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[3] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[4] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[5] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993.

[6] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading". *IEEE Transactions on Computers*, 48(9), 1999.

[7] P. Marcuello, A. González, and J. Tubella. Speculative multithreaded processors. In *12th International Conference on Supercomputing*, 1998.

[8] S. S. Muchnick. *Advanced Compiler Design and Implementation*, chapter 14. Morgan-Kaufmann Publishers, 1997.

[9] K. Ootsu, T. Yokota, T. Ono, and T. Baba. Preliminary evaluation of a binary translation system for multithreaded processors. In *Proceedings of the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, 2002.

[10] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, 1984.

[11] L. Porter, B. Choi, and D. M. Tullsen. Mapping out a path from hardware transactional memory to speculative multithreading. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[12] B. Pottenger and R. Eigenmann. Idiom recognition in the polaris parallelizing compiler. In *Proceedings of the 9th International Conference on Supercomputing*, 1995.

[13] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, 1995.

[14] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[15] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[16] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9), 1999.

[17] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, 1996.

[18] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[19] C. von Praun, L. Ceze, and C. Caşcaval. Implicit parallelism with ordered transactions. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007.

[20] E. Yardımcı and M. Franz. Dynamic parallelization and vectorization of binary executables on hierarchical platforms. *Journal of Instruction-Level Parallelism*, 10, 2008.

[21] W. Zhang, B. Calder, and D. M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[22] W. Zhang, B. Calder, and D. M. Tullsen. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[23] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, 2008.