Appears in the Proceedings of International Conference on Supercomputing (ICS), June 2001

Multiplex: Unifying Conventional and Speculative Thread-Level Parallelism on a Chip Multiprocessor

Chong-Liang Ooi, Seon Wook Kim, Il Park,
Rudolf Eigenmann, Babak Falsafi[‡], and T. N. Vijaykumar

School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907

Electrical and Computer Engineering[‡]
Carnegie Mellon University
Pittsburgh, PA 15213

mux@ecn.purdue.edu, http://www.ece.purdue.edu/~mux

ABSTRACT

Recent proposals for Chip Multiprocessors (CMPs) advocate speculative, or implicit, threading in which the hardware employs prediction to peel off instruction sequences (i.e., implicit threads) from the sequential execution stream and speculatively executes them in parallel on multiple processor cores. These proposals augment a conventional multiprocessor, which employs explicit threading, with the ability to handle implicit threads. Current proposals focus on only implicitly-threaded code sections. This paper identifies, for the first time, the issues in combining explicit and implicit threading. We present the *Multiplex* architecture to combine the two threading models. Multiplex exploits the similarities between implicit and explicit threading, and provides a unified support for the two threading models without additional hardware. Multiplex groups a subset of protocol states in an implicitly-threaded CMP to provide a write-invalidate protocol for explicit threads.

Using a fully-integrated compiler infrastructure for automatic generation of Multiplex code, this paper presents a detailed performance analysis for entire benchmarks, instead of just implicitly-threaded sections, as done in previous papers. We show that neither threading models alone performs consistently better than the other across the benchmarks. A CMP with four dual-issue CPUs achieves a speedup of 1.48 and 2.17 over one dual-issue CPU, using implicit-only and explicit-only threading, respectively. Multiplex matches or outperforms the better of the two threading models for every benchmark, and a four-CPU Multiplex achieves a speedup of 2.63. Our detailed analysis indicates that the dominant overheads in an implicitly-threaded CMP are speculation state overflow due to limited L1 cache capacity, and load imbalance and data dependences in fine-grain threads.

1 INTRODUCTION

Improvements in CMOS fabrication processes continue to increase on-chip integration and transistor count to phenomenal levels. Traditional monolithic superscalar architectures use the increasing transistor counts in extracting instruction-level parallelism (ILP) to achieve high performance. Unfortunately, superscalar architectures are not only becoming less effective in improving the clock speed [23,18,1] and extracting ILP, but are also worsening in design complexity [17] across chip generations. Instead, many researchers and vendors are exploiting the increasing number of transistors to build chip multiprocessors (CMPs) by partitioning a chip into multiple simple ILP cores [10,27,15]. As in traditional multiprocessors, CMPs extract thread-level parallelism (TLP) from programs by running multiple — independent or properly synchronized — program segments, i.e., *threads*, in parallel.

Recent proposals for CMPs advocate speculative, or *implicit*, threading in which the hardware employs prediction to peel off instruction sequences (i.e., *implicit* threads) from the sequential execution stream and speculatively executes them in parallel on multiple cores [27,26,15,28,31,8,30]. Many of the proposals extend a conventional multiprocessor with the ability to handle implicit threads. Conventional multiprocessors employ *explicit* threading, where the software explicitly specifies the partitioning of the program into threads and uses an application programming interface to dispatch and execute threads on multiple cores in parallel. To maintain program correctness, implicitly-threaded architectures rely on the extended multiprocessor hardware to track dependence among threads and verify correct speculation. Upon a misspeculation, the hardware rolls back the system to a state conforming to sequential semantics. To allow proper rollback, implicit threading requires buffering all speculative threads' program state [27]. State-of-the-art cache-based buffering techniques extend hardware cache-coherence protocols to handle speculation [12,16,28,8].

While there are opportunities to exploit both threading models in a single CMP to optimize performance across a wide spectrum of applications, some previous proposals primarily focus on executing the entire programs as implicit threads irrespective of whether there are provably-parallel code sections in a program that can run as explicit threads [12]. Other proposals tacitly assume that the hardware can support explicit threading by default, and focus on evaluating their design for only implicitly-threaded code sections

```
main()
{
      fork(parallel_compute);
      parallel_compute();
      .....
      serial_compute();
      .....
}
parallel_compute()
{
   for(i=id*N; i<id*(N+1); i++)
      A[i] = A[i] + X;
   barrier();
   for(i=id*N; i<id*(N+1); i++)
          ... = A[2] + Y;
}
```

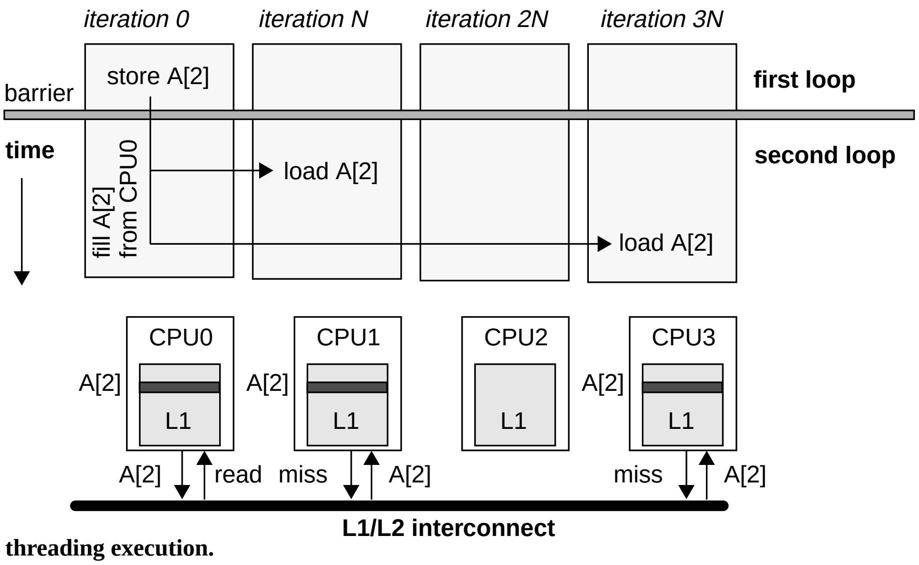

FIGURE 1: An example of explicit threading execution.

[16,28,8]. This paper takes the *first step towards evaluating a hybrid architecture* to combine the two threading models, and presents a detailed analysis of execution overheads and compiler trade-offs given the choice of two threading models for analyzable code sections.

We propose the *Multiplex* architecture to capitalize on the similarities among the hardware resources in explicitly-threaded and implicitly-threaded CMPs and unify support for both threading models in a single CMP. We have developed a fully-integrated compiler to automate code generation for implicit, explicit, and Multiplex CMPs. The main contributions of this paper are:

- Using the compiler and cycle-accurate simulation we show that neither threading architecture alone performs consistently better than the other across the benchmarks. A CMP with four dual-issue CPUs on average achieves a speedup (over a single dual-issue CPU) of 1.48 and 2.17 using implicit-only and explicit-only threading respectively. Multiplex matches or outperforms the better of the two CMPs for every benchmark and, on average, achieves a speedup of 2.63;
- We present the *Multiplex Unified Coherence and Speculation (MUCS)* protocol which provides unified support for explicit and implicit threads in a single application, without additional hardware. MUCS groups a subset of protocol states in a previously-proposed protocol for an implicitly-threaded CMP [12] to provide a state-of-the-art write-invalidate protocol for explicit threads;
- We present a detailed execution time breakdown analysis of the entire programs including both implicit and explicit sections. Our results indicate that the dominant overheads in an implicitly-threaded CMP are speculation state overflow due to limited L1 cache capacity, and load imbalance and data dependence overhead in fine-grain threads;
- Parallel code sections can be transformed into either implicit or explicit threads. We present evidence that a naive thread selection of explicit threads whenever possible would lead to inferior performance due to high explicit thread dispatch overhead in fine-grain threads.

In the following section, we describe current explicit and implicit architectures. In Section 3, we introduce Multiplex. Section 4 presents a discussion of key factors effecting performance in the two architectures. Section 5 presents the simulation methodology and results. Section 6 presents a summary of related work. Finally, Section 7 presents a summary and concludes the paper.

2 BACKGROUND: EXECUTING EXPLICIT & IMPLICIT THREADS

In this section, we briefly describe and provide examples of thread execution and the required hardware support in explicitly-threaded and implicit-threaded CMPs. While there are key differences in thread dispatch, execution, and communication in these CMPs, the dominant fraction of hardware resources required — e.g., CPU cores and shared memory — is similar and can be exploited by a CMP supporting both models.

In explicitly-threaded architectures, the application software either *eliminates* data dependence through advanced parallelizing compiler analysis and techniques, or *specifies* every instance of data dependence using a synchronization primitive (e.g., a barrier). Moreover, software specifies inter-thread control dependence using a thread dispatch primitive (e.g., a fork call).

Figure 1 shows a simple example of a program running on an explicitly-threaded CMP. In this example, the main thread executes sequentially (not shown) on CPU0, and forks parallel explicit child threads so that each CPU executes the function *parallel_compute* simultaneously. The function includes a pair of loops, where each thread executes a fraction of the loop iterations. The first loop computes and writes to array *A*. In the second loop, every loop iteration is dependent on the value of *A[2]* created by CPU0 in the first loop and stored in its L1. The CPUs' L1 cache controllers imple-

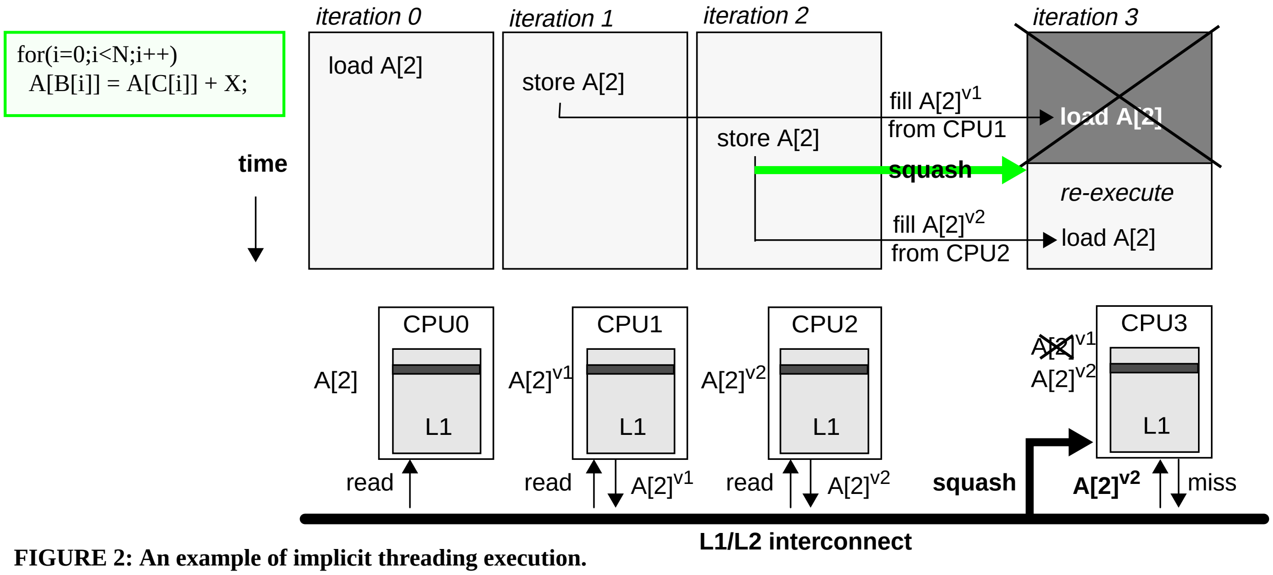

FIGURE 2: An example of implicit threading execution.

ment a snoop cache-coherence protocol which identifies *A[2]*'s most recent copy to be in CPU0's L1, and copies it into other CPUs' L1s (e.g., CPU1 and CPU3) on demand.

In contrast, in implicitly-threaded architectures, inter-thread data and control dependences are *implicit* in the sequential program order. The hardware infers the existence of data dependence — e.g., between a memory read instruction and a preceding program-order memory write instruction to the same location. Similarly, the hardware resolves inter-thread control flow dependence. Because hardware receives no information from software, it relies on dependence prediction and speculation techniques to guess the missing information and deliver high performance.

Figure 2 shows a simple example of a program running on an implicit-threaded CMP. In this example, a loop computes over array *A* with loop iterations that have unknown dependences at compile time. A compiler for an implicitly-threaded architecture (e.g., the Multiscalar compiler [33]) partitions the loop and assigns each implicit thread a single loop iteration. Unlike explicitly-threaded architectures, implicitly-threaded architectures rely on hardware prediction to dispatch threads. As shown in the example, the predictor selects and dispatches subsequent loop iterations, starting from iteration 0, on the CPUs in cyclic order. Because iteration 0 is the "oldest" thread executing in program order, it is guaranteed to complete and is said to be "non-speculative". Dispatch prediction for a thread is only verified when all preceding threads complete, therefore all threads except for iteration 0 are "speculative" and may be "squashed" if mispredicted.

Assume that iterations 0, 1, 2, and 3 access the same element *A[2]*. Upon missing on a load from *A[2]*, CPU0's thread obtains a copy of the corresponding cache block from L2 and marks the block as non-speculative. After a few cycles, CPU1's thread (i.e., the speculative iteration 1) misses on a store to *A[2]*, and the protocol supplies a copy of the block from L2. CPU1 then creates a speculatively renamed version of the block, denoted by $A[2]^{v1}$, without invalidating CPU0's copy (as would be done in explicitly-threaded architectures), and marks the block as speculative dirty. When CPU3's thread misses on a load from *A[2]*, the protocol supplies CPU1's version of the block, $A[2]^{v1}$, because CPU1 is the closest preceding thread, and CPU3 marks its own copy as speculatively loaded.

Next, CPU2 misses on a store to *A[2]*, it creates yet another speculative renamed version of the block, $A[2]^{v2}$, without invalidating $A[2]^{v1}$. The protocol subsequently squashes CPU3 (and any future threads) because CPU3 prematurely loaded CPU1's version, $A[2]^{v1}$, instead of the sequentially correct CPU2's version, $A[2]^{v2}$. Squashing CPU3 also invalidates the blocks speculatively accessed by CPU3. The protocol maintains the program order between CPU1's and CPU2's versions, as part of the protocol state to provide the correct version for future accesses to *A[2]*. CPU3 re-executes and loads $A[2]^{v2}$ from CPU2.

Upon completion, the threads "commit" in sequential order, marking the speculatively accessed blocks as non-speculative (or committed). Because all future iterations access different elements of *A*, cache blocks accessed in those iterations are first marked as speculative, and then committed without causing any squashes. Because the L1 caches maintain the program order among all data assesses (for both loads and stores) to track dependences and guarantee correct execution, speculative data are not allowed to leave the caches; any capacity and conflict problems causing a speculative block replacement stall the CPU until it becomes non-speculative, resulting in substantial performance loss.

3 MULTIPLEX: UNIFYING EXPLICIT/ IMPLICIT THREADING

In this paper, we propose *Multiplex*, an architecture that unifies explicit and implicit threading on a chip multiprocessor. Multiplex obviates the need for serializing unanalyzable program segments by using implicit threading's speculative parallelization. Multiplex avoids implicit threading's speculation overhead and performance loss in compiler-analyzable program segments by using explicit threads. Multiplex capitalizes on the similarities between implicit and explicit CMP hardware to support both threading models on the same set of execution cores with minimal modifications to an implicit CMP.

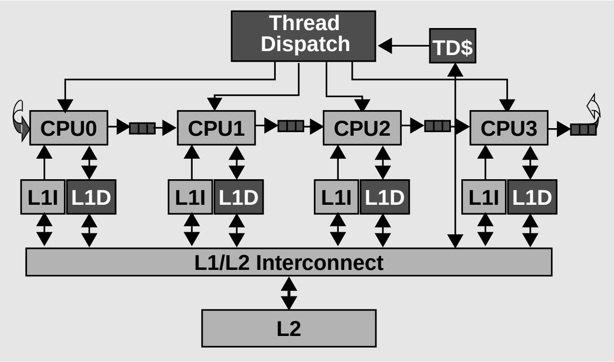

FIGURE 3: A Multiplex CMP. The figure depicts the anatomy of a Multiplex CMP. The blocks appearing in a light shade of gray are components used in a conventional (explicitly-threaded) multiprocessor including the processing units, the L1 instruction caches, the system interconnect, and the L2 cache. The blocks appearing in a dark shade of gray are components enabling implicit and hybrid implicit/explicit threading in Multiplex including the thread dispatch unit, the thread descriptor cache (TD$), the L1 data caches, and the register communication mechanism.

The key mechanisms required for a threading model are: (1) *thread selection*, a mechanism to partition the code into distinct instruction sequences, (2) *thread dispatch*, a mechanism to assign a thread to execute on a CPU, and (3) *data communication and speculation*, mechanisms to propagate data (i.e., register and memory) values among independent threads, to allow implicit threads to privatize data in multiple caches under the same memory address, and to guarantee correct program execution. In the following subsections, we present hardware and compiler mechanisms for thread selection and dispatch, and data communication and speculation to unify explicit and implicit threading within a single application.

Figure 3 illustrates a Multiplex CMP. Our Multiplex CMP is loosely derived from the Wisconsin Multiscalar [27,12]. As in traditional small-scale multiprocessors, a Multiplex CMP includes a small number of conventional CPU cores with first-level instruction and data caches and a shared level-two cache [21]. As in Multiscalar, Multiplex includes support for speculative thread dispatch consisting of a dispatch unit and a thread descriptor cache; register communication queues; and memory communication, speculation, and disambiguation through level-one data caches. Multiplex unifies cache coherence with memory renaming and disambiguation in level-one caches through a single snoopy bus protocol.

3.1 Thread Selection

Multiplex relies on a unified compiler infrastructure to generate both explicit and implicit threads. Unlike state-of-the-art compilers which are limited to compiling for a specific threading model, in Multiplex the compiler has the opportunity to choose between two threading models to maximize performance on a per program and per program segment basis. The choice between threading models depends on program and system characteristics.

To minimize execution overhead due to speculation, the Multiplex compiler always searches first for statically parallelizable program segments and partitions profitable sections into *explicit threads*. Explicit threads maximize the parallelism exploited, minimize the parallelization overhead by selecting coarse-grain threads, eliminate speculation overhead, and realize the raw hardware speeds of multiple CMP cores. Multiplex relies on a state-of-the-art parallelizing compiler to analyze programs and generate explicit threads. These compilers (e.g., Polaris [4], SUIF [14]) use a myriad of techniques to test [5,25,11] and eliminate data dependence in program segments [2,32,24,13]. Moreover, these compilers increase thread performance in analyzable program segments through code transformations to optimize for memory hierarchy locality and communication latency [14].

Once the compiler selects appropriate program segments to execute as explicit threads, the compiler chooses the rest of the program as *implicit threads*. These program segments typically consist of those with control flow or data dependences that are unanalyzable at compile time. Multiplex extracts parallelism from implicit threads at runtime with the help of hardware speculation. Unlike explicit threading where software invokes thread dispatch using an application-programming interface, in implicit threading the software merely specifies thread boundaries and not the control flow among them [33]. The hardware in turn predicts and speculatively dispatches threads at runtime to maintain instruction execution flow in accordance with the sequential execution semantics. Multiplex also relies on the compiler to generate implicit threads, and thereby benefits from many key transformation techniques available at compile time to improve implicit thread performance [33]. Implicit threads are typically fine-grain so as to minimize the likelihood of speculative state overflow.

Key advantages to unifying thread selection. There are scenarios in which there is a trade-off between the two threading models for statically parallelizable programs. Loops with small bodies that iterate for a small number of times are best executed as implicit threads due to the high explicit dispatch overhead and low implicit data speculation overhead. Moreover, program segments that are not evenly partitionable into thread numbers that are multiples of CPUs will result in a significant load imbalance if executed entirely as explicit threads. The compiler can peel off the tail part of such a program segment and execute it in parallel with subsequent program segments as implicit threads to eliminate the load imbalance. The compiler's flexibility in choosing the threading model helps complement the strengths of both models, thereby improving application performance. This issue will be discussed in further detail in Section 5

3.2 Thread Dispatch

In Multiplex, dispatching a thread on a CPU involves: (1) assigning a program counter to the CPU indicating the address of the first instruction belonging to the thread, (2) assigning a private stack pointer to the CPU, and (3) implementing a dispatch "copy" semantics copying the stack and register values prior to the dispatch to all dispatched threads; as in conventional threading models, Multiplex uses a single address space for all the threads and requires copy semantics only for stacks and registers (and not memory) upon dispatch.

To minimize thread dispatch overhead, Multiplex supports explicit thread dispatch directly in the instruction set. A `fork` instruction takes an argument in an architectural register, and assigns it to the program counter of all other CPUs. Once dispatched, threads proceed until the execution reaches a `stop` instruction. Upon thread completion, an application may dispatch new threads through subsequent executions of the `fork` instruction. The Multiplex system initialization library allocates private stacks for all CPUs, and uses a `setsp` instruction to assign the pre-allocated stacks to individual CPUs. Because fork requires a register "copy" semantics, the library call forks threads indirectly through a "wrapper" procedure that copies the necessary register into individual CPU stacks prior to invoking thread code.

Multiplex dispatches implicit threads in program order [27]. The thread dispatch unit (Figure 3) uses the current implicit thread to predict and dispatch a subsequent thread. Threads dispatch, execute, and commit sequentially. A control or data dependence violation results in squashing threads in program order, and redispatching threads. A thread descriptor embedded in the code prior to the thread code fragment includes addresses of possible subsequent dispatch "target" threads. The thread dispatch unit includes a thread predictor that selects one of the target threads to dispatch. The thread descriptor also includes the information necessary to identify register values a thread depends which must be communicated from previously dispatched threads [27]. To accelerate thread dispatch, a thread descriptor cache (Figure 3) caches recently referenced thread descriptors.

Selecting and switching dispatch mode. Multiplex selects the threading "mode" (i.e., implicit or explicit) at dispatch time. In Multiplex, the compiler includes a mode bit (set for explicit threads and reset for implicit threads) in every thread descriptor indicating the threading mode. The hardware maintains a global mode bit indicating the current mode. As a first step towards implementing hardware to support both threading modes, we preclude overlapping execution of threads from different modes. Therefore, upon switching from executing implicit threads to explicit threads, Multiplex suspends thread dispatch until all speculation completes and all threads commit. Upon switching from explicit to implicit threads, Multiplex suspends dispatch until all instructions from all explicit threads have committed.

3.3 The MUCS Protocol

Much like all modern architectures, Multiplex uses registers and memory to store program state. As in Multiscalar, implicit threads in Multiplex share *both* register and memory state among each other. Explicit threads, however, execute as in shared-memory multiprocessors and only share memory and not register state. Because register communication in Multiplex is identical to that in Multiscalar, we do not discuss register communication any further and refer the reader to [6,27]. In this section we focus on memory data communication and speculation among threads.

Both explicit CMPs and proposals for scalable implicit CMPs [21,28,12] rely on snoopy bus-based protocols to maintain memory data integrity. In both CMPs, the CPUs' private caches enable efficient data sharing by making copies of accessed data close to each CPU. The main responsibility of the memory system is to track the copies so that the correct copy is delivered upon a memory access. In explicit CMPs, the memory system locates the most-recently written copy either from main memory (if there is no cached dirty copy) or from another cache (if it has a dirty copy) for loads, and ensures that no stale copies exist in other caches (e.g., through invalidations or updates) for stores.

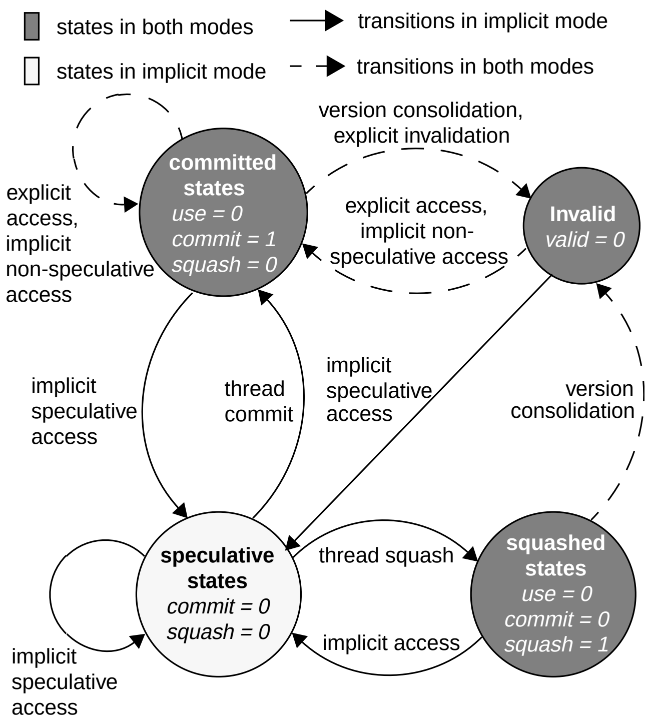
states in both modes
states in implicit mode
transitions in implicit mode
transitions in both modes
version consolidation, explicit invalidation
explicit access, implicit non-speculative access
committed states
use = 0
commit = 1
squash = 0
explicit access, implicit non-speculative access
Invalid
valid = 0
implicit speculative access
thread commit
implicit speculative access
version consolidation
speculative states
commit = 0
squash = 0
thread squash
implicit access
squashed states
use = 0
commit = 0
squash = 1
implicit speculative access

FIGURE 4: A high-level state transition diagram for MUCS

In implicit CMPs, the memory system provides similar support but in the presence of speculative loads and stores. As in explicit CMPs, in implicit CMPs multiple processors may read-share data and maintain multiple copies which the protocol will guarantee to be coherent. Unlike explicit CMPs, implicit CMPs also allow processors to maintain multiple *versions* of data to co-exist to respect the sequential program order of memory accesses. The memory system tracks loads to detect (and squash) any load that prematurely accesses a location before a prior (program-order) store is complete. The memory system also creates a new version for every store and tracks the program order among the multiple versions.

Multiplex unifies coherence and versioning into a single protocol called the *Multiplex Unified Coherence and Speculation* (MUCS) protocol. MUCS uses a finite-state machine with two sets of state: (1) states which implement coherence among multiple copies of (a single version of) data in both implicit and explicit machine modes, and (2) states which implement speculation to maintain program order among multiple versions in the implicit mode.

MUCS is derived from SVC [12], a speculative versioning protocol for an implicit CMP. As in SVC, the key design objective for MUCS is to minimize speculation overhead in two respects. First, dependence resolution in the common case should be handled within the cache, minimizing the frequency of bus transactions. Second, thread commits/squashes should only require *en masse* cache operations and preclude examining individual cache blocks to update state. The exact details of the SVC protocol state and transitions are published in [12]. In the following, we briefly describe the protocol states and transitions in MUCS and show

Table 1: MUCS protocol state and actions.

State bit	Action
use	set per access in implicit mode by speculative loads executed before a store; used only in implicit mode to flag premature loads violating store-to-load order; cleared for the entire cache en masse upon thread commit or squash in implicit mode
dirty	set by all stores in both modes; used to write back a version on invalidation in explicit mode and version consolidation in both modes; cleared on write back to next level in both modes
commit	set for the entire cache en masse at thread commit in implicit mode and set per access in explicit mode; used in both modes to allow replacements of committed dirty versions; cleared on every access in implicit mode
stale	set only in implicit mode on store miss from a succeeding CPU with a potentially more recent version, and by cache fills if a succeeding CPU has an uncommitted/unsquashed dirty versions; used in both modes to force misses (if commit bit set), and to consolidate the most recent committed version among multiple committed/squashed versions of a previous cyclic order; cleared in both modes for the consolidated version
squash	set for the entire cache en masse upon thread squash in implicit mode; used in both modes to force misses on the next access to the block; commit bit has precedence over squash bit; cleared on every access in implicit mode, and in both modes for the consolidated version
valid	set per cache fill on cache misses in both modes; used in both modes to determine validity of tag (not data), and allow replacements; cleared on invalidation in explicit mode, and in both modes for all committed/squashed versions other than the consolidated version

how MUCS derives a write-invalidate coherence protocol for the explicit mode by simply adding protocol transitions to SVC.

Figure 4 shows a high-level state transition diagram for MUCS. The figure depicts a breakdown of states used in implicit mode only, and states shared between the implicit and explicit modes. The *committed states* implement a state-of-the-art write-invalidate protocol actively used in the explicit mode. The *squashed states* correspond to blocks accessed speculatively by (prior) squashed implicit threads and left behind in those states. The *speculative states* are those states corresponding to references by speculative implicit threads. A cache block can only be placed in squashed or speculative states by implicit threads. An attempt to evict a block in a speculative state suspends execution on a CPU until all prior threads commit, and the current thread becomes the head of the ring. *Invalid* corresponds to an invalidated block in either mode.

Table 1 summarizes the protocol state bits and semantics in implicit and explicit modes. A missing load will cause a cache fill from the latest version of the cache block in program order. The *use bit* indicates a speculative load by a thread so that a store by preceding (program-order) threads can detect dependence violation. A store from an earlier thread squashes a future thread (and all its successors) that has the use bit set for the block.

The *dirty bit* records all modified blocks — i.e., blocks that have been stored to. When a store creates a new version, the block in preceding CPUs are marked potentially stale with the *stale bit*. Such blocks are only potentially stale because the new version itself is speculative and may be squashed later. While the use bit clearly distinguishes all speculatively-loaded blocks, the dirty bit is set for all stored blocks whether speculative or non-speculative. MUCS sets the *commit bit* on thread commits and clears the bit (i.e., commit bit cleared indicates speculative) on accesses from speculative threads. When a thread commits, all the commit bits in the entire cache are set en masse.

To avoid scanning the cache for invalidating the blocks touched by a squashed thread, MUCS uses a *squash bit*. When a thread is squashed, the squash bits in the entire cache are all set en masse. MUCS also clears all the use bits en masse on a thread squash. Access to a block with the squash bit set forces a miss and clears the squash bit.

MUCS uses the cyclic thread dispatch order among the CPUs to infer the program order among multiple versions. Unfortunately, it does not suffice to solely rely on the cyclic order to determine program order because the position of the oldest thread in the system rotates from one CPU to the next (as threads commit). To avoid this problem, MUCS *consolidates* the versions from a previous cyclic order at the next access. Versions from a previous order are guaranteed to be committed or squashed. MUCS locates and writes back the most recent committed dirty version and invalidates all the other committed/squashed versions.

Switching between implicit and explicit modes. Because explicit threads may access data which was last accessed by an implicit thread, explicit threads may encounter blocks with squash or stale bit set. MUCS treats these explicit mode accesses as misses, consolidates the most recent committed version via a bus snoop, and supplies the consolidated version to the requesting CPU. Because explicit mode accesses are indistinguishable from committed implicit mode accesses, there is no overhead for switching from implicit to explicit mode, or vice versa.

Much as other conventional coherence protocols, MUCS can employ optimizations such as exclusive caching and/or snarfing [9]. These optimizations have been employed by SVC and are applicable to MUCS as well. Much like other speculative protocols [12,29,16,8], MUCS can optimize away false squashes by maintaining protocol state at finer (e.g., word) than cache block granularity. In this paper, we take the first step towards unifying implicit and explicit threading and do not explore the granularity issue.

4 KEY PERFORMANCE FACTORS

Multiplex combines the performance advantages of explicit and implicit threading models. There are key factors affecting the performance of either model. Therefore, to gain insight on Multiplex's performance, we qualitatively evaluate these factors in this section. In Section 5, we present simulation results that corroborate our intuition from this discussion.

Thread size. Thread size is a key factor affecting performance in both explicit-only and implicit-only CMPs. Larger threads help (1) increase the scope of parallelism, and may help reduce the likelihood of data dependence across threads, and (2) reduce the impact

Table 2: System configuration parameters.

Processing Units	
CPUs	4 dual-issue, out-of-order
L1 i-cache	8K, 2-way, 1 cycle hit
L1 d-cache	8K, Direct Map, 16-byte block, 1-cycle hit, byte-level disambiguation
Squash buffer size	64 entries
Reorder buffer size	32 entries
LSQ size	32 entries
Functional units	3 integer, 1 floating-point, 1 memory
Branch predictor	path-based, 4 targets
System	
Thread Predictor	path-based, 2 targets
Descriptor Cache	16K, 2-way, 1-cycle hit
Shared L2	2M, 8-way, 64-byte block, 9-cycle hit and transfer
L1/L2 interconnect	snoopy split-transaction bus, 32-bit wide
Memory latency	80 cycles

Table 3: Applications and input sets.
^{S} indicates scaled down number of loop iterations in the interest of reduced simulation time.

name	input	#of inst (billions)
SPECfp95 Benchmarks		
fpppp	train	0.470
apsi	train	2.847
turb3d	train^S	0.332
applu	train	0.649
wave5	train^S	0.114
su2cor	test	1.114
tomcatv	test	0.440
hydro2d	test	1.141
swim	test	0.753
mgrid	train^S	2.810
Perfect Benchmarks		
flo52	std	3.466
arc2d	std^S	1.530
trfd	std	3.405

of thread dispatch/completion overhead. Larger threads, however, increase the likelihood of speculative state overflow — i.e., overflow of blocks in the speculative state due to the limited capacity of L1 caches — in implicit-only CMPs, by increasing the required storage to maintain speculatively produced data.

Load imbalance. A key shortcoming of explicit-only CMPs is their inability to exploit parallelism in program segments that are not analyzable at compile time. Unfortunately, even a small degree of unknown dependences prevents a parallelizing compiler from generating explicit threads, resulting in a serial program segment. Explicit-only CMPs' performance depends on the fraction of overall execution time taken by the serial program segments. Multiplex can execute the serial program segments as implicit threads, significantly improving performance over explicit-only CMPs in programs with large serial segments.

In implicit-only CMPs, load imbalance is highly dependent on the control flow regularity across threads. For instance, inner loops with many input-dependent conditional statements may result in a significant load imbalance across the CPUs. Explicit-only CMPs use coarse-grain threads in which control flow irregularities across basic blocks *within* a thread often have a cancelling effect, reducing the overall load imbalance across threads. Control flow irregularities only impact performance in Multiplex for program segments that execute as implicit threads.

Data dependence. Parallelizing compilers can often eliminate known data dependences (e.g., through privatization or reduction optimization). Unknown data dependences, however, result in serial program segments in explicit-only CMPs, reducing performance. Using fine-grain threads in implicit-only CMPs often causes high data dependence and communication across adjacent threads. Data dependence contributes to threading overhead because a dependent thread must at a minimum wait for data to be produced. While dependences through registers are synchronized, memory dependences may incur additional speculation overhead when memory synchronization hardware is unable to prevent unwanted speculation [20]. Multiplex increases opportunity for eliminating thread dependence by executing compile time analyzable program segments as explicit threads.

Thread dispatch/completion overhead. Thread dispatch/completion overhead only plays a major role for fine-grain threads, where the overhead accounts for a large fraction of thread execution time. In explicit-only CMPs, thread dispatch incurs the overhead of copying of stack parameters and register values. Thread completion incurs the overhead of flushing the CPU load/store queues to make memory modifications visible to the system. Explicit-only CMPs, however, use fine-grain threads only when the compiler can not analyze dependences among larger thread bodies. Multiplex can execute such fine-grain threads as implicit threads, thereby reducing the thread dispatch/completion overhead.

Speculative state overflow. Data speculation in implicit-only CMPs is limited by the amount of buffering data caches can provide. Speculation requires buffering all versions, causing data caches to fill up quickly and overflow for memory-intensive threads and/or long-running threads. Because, speculative data are not allowed to leave the caches, execution for a speculative thread overflowing in the cache stops until all prior threads commit and the thread becomes non-speculative. While data speculation is always performed in implicit-only threads, threads are not always data-dependent. Multiplex significantly reduces the data specula-

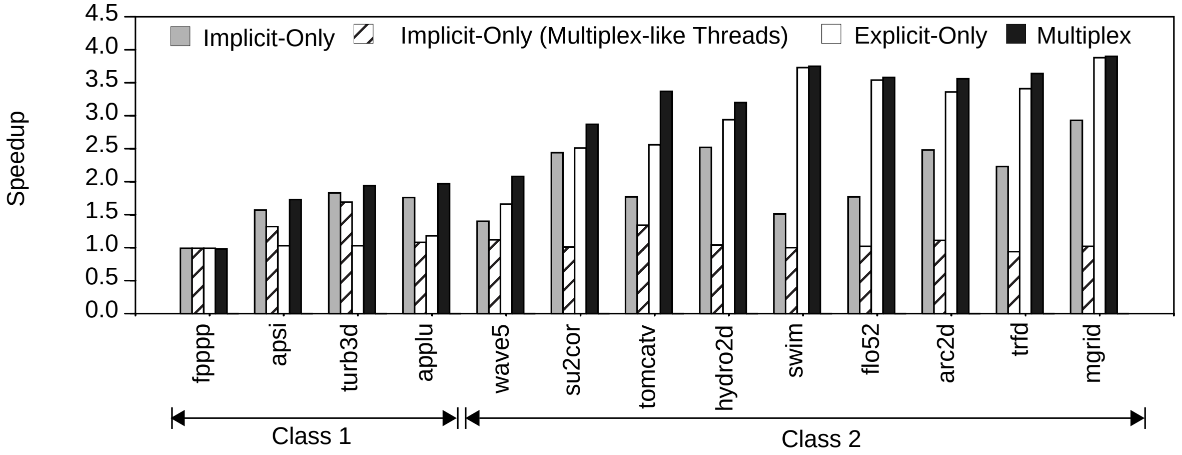

FIGURE 5: Performance of Multiplex, implicit-only, and explicit-only CMPs. In class 1 applications, implicit-only outperforms explicit-only and vice versa in class 2 applications. In all applications, Multiplex matches or exceeds the performance of the better alternative. Choosing larger threads for implicit-only execution always performs worse.

tion overhead by execution independent threads as explicit threads, obviating the need for speculation.

5 PERFORMANCE EVALUATION

In this section, we quantitatively evaluate a Multiplex CMP's performance and compare it against that of an "implicit-only" and "explicit-only" CMP using simulation. We first give a brief overview of our compiler infrastructure, the experimental methodology, and the application and input parameters we use. Next we present the following results: (1) Multiplex performs as well or better than the best of implicit-only or explicit-only CMPs by allowing optimal threading mode selection within and across applications, (2) implicit-only CMPs' key sources of overhead are limited capacity in L1 caches to maintain speculative data, and load-imbalance and data dependence in fine-grain threads, (3) selecting larger threads may help remove data dependences and alleviate load imbalance in fine-grain implicit threads but prohibitively increase the likelihood of speculative state overflow, and (4) naive selection of provably-parallel code as explicit threads is suboptimal for fine-grain threads with high thread dispatch overhead.

5.1 Methodology and Infrastructure

We have developed a cycle-accurate simulator of a Multiplex CMP. Our simulator models multiple ILP CPU cores and pipelines, the memory hierarchy, and an implementation of the Multiplex threading mechanisms in detail. Table 2 summarizes the processor and system configuration parameters we use in this study. The CMP includes four dual-issue out-of-order cores, each with L1 instruction and data caches, backed up by an L2 cache. The simulator models the thread dispatch unit, descriptor cache, and the register communication queues for the implicit mode, the dispatch and synchronization instructions for the explicit modes, and the MUCS protocol.

Our compiler infrastructure integrates Polaris [4], a state-of-the-art parallelizing preprocessor generating explicit threads, with the Multiscalar compiler [33], a gcc-based compiler for generating implicit threads. Our compiler infrastructure is fully automated, except for the selection of explicit versus implicit threads, which is done semi-automatically (discussed in Section 5.4). We compile the benchmarks as is, without modifying the source code. To evaluate and compare Multiplex against explicit-only and implicit-only architectures, the compiler allows for generating implicit-only and explicit-only threads when compiling applications.

We use a combination of benchmarks from the SPECfp95 [7] and the Perfect [3] suites. Table 3 shows the benchmarks, the used input data sets and the number of instructions executed for each benchmark. In the interest of simulation turnaround time, we scale down the number of outer loop iterations for some of the applications. This change in input set has a minimal impact on our performance results since the inherent communication/computation characteristics of the applications remain the same. We do not evaluate integer benchmarks because our compiler infrastructure can parallelize only Fortran programs.

5.2 Base Case Results

Figure 5 compares speedups for the Multiplex CMP against the explicit-only and the implicit-only CMPs. We measure speedup relative to a superscalar processor configured identically as one of Multiplex' CPUs. Because Multiplex differs from an implicit-only architecture in both the compiler and the architecture, the figure also shows, as a reference point, the performance of the implicit-only CMP with a compilation scheme close to the one for Multiplex. This scheme selects threads from outer parallel loop instead of innermost loops.

The figure divides the applications into two classes: class 1 applications favor the implicit-only CMP and class 2 applications favor the explicit-only CMP. The results indicate that there is a significant performance disparity between the explicit-only and the implicit-only CMPs across the applications. In class 1 applications, the implicit-only CMP achieves on average 42% higher speedups and at best 78% higher speedups than the explicit-only CMP. In contrast, in class 2 applications the explicit-only CMP achieves on average 45% higher speedups and at best 147% higher speedups than the implicit-only CMP.

Multiplex always performs best. In all applications, Multiplex makes the correct choice between the explicit and implicit threading models, always selecting the better of the two. Multiplex on

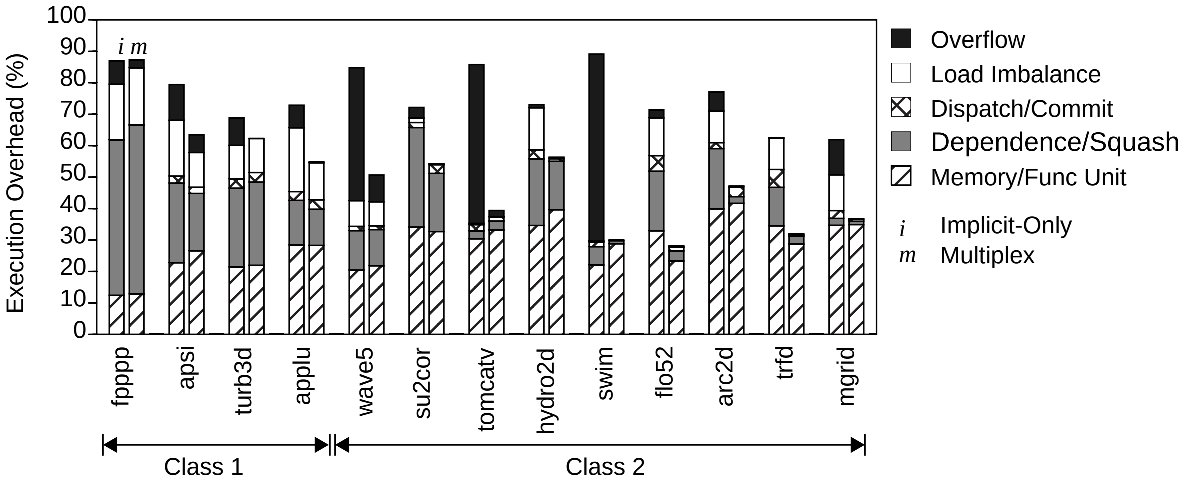

FIGURE 6: Overheads of the Implicit-only and the Multiplex architecture. For class 1 applications Multiplex substantially reduces the speculative state overflow. For class 2 benchmarks, Multiplex virtually eliminates the threading-related overheads in most of the applications. Overheads not related to threading, indicated by memory and functional unit stalls, remain nearly constant.

average achieves a speedup of 2.63, improving speedups by 21% over explicit-only and 41% over implicit-only CMPs. In seven applications, Multiplex improves speedups over the better of the two on average by 10%. To better understand application performance on each architecture, we evaluate the key factors affecting performance in the next section.

The implicit-only performance with the large-thread selection scheme used in Multiplex is always worse than with the basic scheme. These results show that it is not possible to improve implicit-only performance by simply selecting threads from compiler-recognized parallel loops, as is done in Multiplex. We will discuss this further in Section 5.6. In the following figures we use the better of the two performance numbers as a reference point.

5.3 Detailed Overhead Analysis

Table 4 shows the percentage of the original (serial) execution of each application that can be recognized as parallel by the compiler. The opportunity for explicit-only architectures is to execute this fraction of the application in parallel. Amdahl’s law dictates that a substantial fraction of serial execution can offset the gains from parallelism and severely limit overall performance. For example, in *su2cor*, parallelizing 81% of the application limits speedups to at most 2.5 (i.e., 1/(0.8/4+0.2)=2.5). This is a key source of performance degradation in explicit-only CMPs, which Multiplex can overcome through executing the serial sections as implicit threads.

Figure 6 shows the execution overheads of the implicit-only and the Multiplex CMPs. The figure plots overhead (i.e., the number of processor cycles not contributing to computation) as a fraction of overall execution time of the implicit-only CMP. The figure shows both overheads due to threading mechanisms and overheads intrinsic to the base superscalar cores. Threading-related overheads include speculative state overflow (processor stalls because it cannot replace a *speculative* cache line), load imbalance (uneven workload of the processor cores), dispatch/commit (additional code executed for thread management), and dependence/squash (wait time or roll-backs due to dependences). Superscalar-related overheads include pipeline hazards and memory stalls. They are nearly identical in the two architectures. In the following discussion we will concentrate on threading-related overheads.

We will first consider overheads in the implicit-only CMP and then discuss the changes when going to Multiplex. The figure shows that a key source of overhead is data dependences and squashes. Our measurements indicate that squash overhead is small in all cases, except in *fpppp*. The thread predictor exhibits high prediction accuracies for all the applications because loop branches (at the thread boundaries) are typically predictable. The memory dependence hardware (i.e., the squash buffer[20]) can also synchronize most dependences because most implicit threads are fine grain with small instruction footprints. The squashes in *fpppp* are due to low hit rates in the squash buffer because of *fpppp*’s large threads [20].

Load imbalance is another key overhead factor. *Fpppp*, *apsi*, *turb3d*, *applu*, and *wave5* have control flow irregularities in the inner loops (i.e., loop iterations including input-dependent conditionals [33]). Because the implicit-only CMP uses fine-grain threads, it primarily targets inner loops and therefore can suffer from load imbalance in these applications. Similarly, thread completion overhead of flushing the load/store queues is considerable in some applications due to the small thread size. Load imbalance can further result from stalls due to speculative state overflow.

Table 4: Fraction of execution time of each application that is provably-parallel by the compiler and converted to explicit threads.

Benchmark	*fpppp*	*apsi*	*trub3d*	*applu*	*wave5*	*su2cor*	*tomcatv*	*hydro2d*	*swim*	*flo52*	*arc2d*	*trfd*	*mgrid*
Fraction Threaded (%)	0	72	34	97	70	81	82	77	99	93	95	100	95

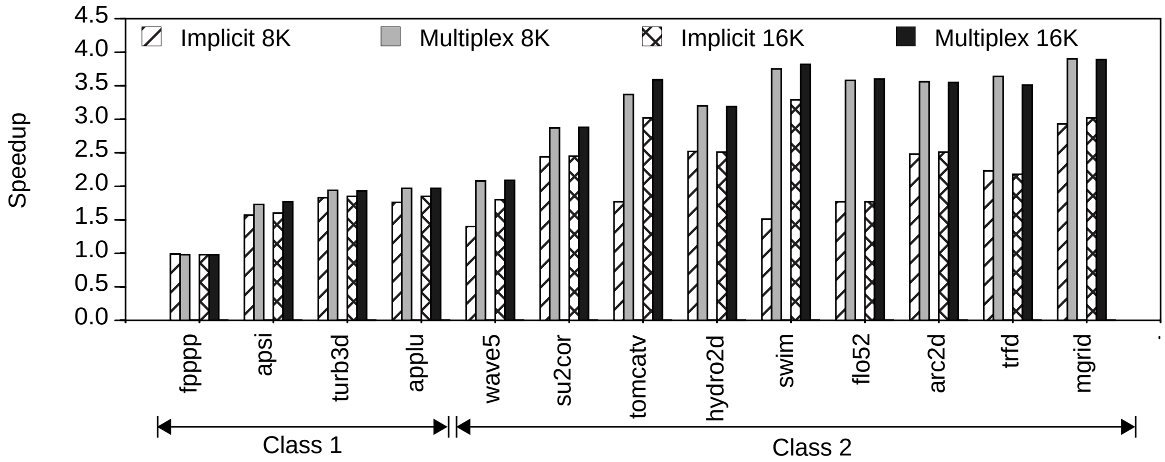

FIGURE 7: Impact of cache size. While the Multiplex performance is nearly unaffected by the change in cache size, the change impacts the implicit-only CMP's performance significantly in *wave5, tomcatv,* and *swim*. The caches are all direct-mapped.

Finally, data speculation overhead due to speculative state overflow is substantial in the implicit-only CMP for most applications, but dominates in *wave5*, *tomcatv,* and *swim*. Our current compiler techniques attempt to select thread sizes to minimize the speculative state overflow [33]. However, in Section 5.6 we show that using larger threads to increase parallelism and eliminate dependences would prohibitively increase the speculation overhead for most applications. This overhead is one of the key limitations of implicit-only architectures and a motivation for Multiplex.

Figure 6 also indicates that Multiplex reduces much of the overheads in the implicit-only CMP. Multiplex exploits advanced parallelization techniques to eliminate data dependences and generate coarse-grain explicit threads (from iterations of outer loops), significantly improving performance over the implicit-only CMP. For class 1 applications, the impact of these transformations on dependence/squash overheads is minor. However, in *turb3d* and *applu* speculative state overflow has disappeared. The reduction of overheads in class 2 applications is very significant. Overflow is nearly eliminated. Load imbalance has disappeared in all cases but *wave5*, and dependence-related overheads are significantly reduced in six out of the nine applications.

5.4 Selecting Implicit vs. Explicit Threads

Combining explicit and implicit threading schemes is non-trivial. A compiler algorithm that selects explicit threads whenever it detects parallelism would lead to inferior performance, due to the overhead for generating and dispatching explicit threads. Two of the class 1 applications, *apsi* and *applu* suffer from a small thread size. Table 5 shows that in these applications the eager explicit thread selection scheme would degrade performance by 41%. We have also developed a conservative algorithm that generates implicit threads for innermost loops, even if they are found to be fully parallel. The conservative algorithm improves performance of *apsi*, *applu* and *turb3d* to the level shown in Figure 5. However it would decrease performance in *tomcatv* because its innermost loops are large and thus would incur speculative state overflow. For the measurements in Figure 5 we have chosen the suitable algorithm by compiler options on an application by application basis. The need for developing accurate compiler performance prediction schemes that trade-off these overheads is an important finding of our work.

5.5 Impact of Cache Size

Figure 7 shows the performance impact of increasing the cache size. We have measured two cache configurations, 8K direct-mapped and 16K direct-mapped, for all the CMPs. The results show a significant increase in performance when selecting a large cache size for *wave5, tomcatv,* and *swim*. This increase also indicates that the performance would decrease substantially when increasing the data set of the applications. This decrease is due to the speculative state overflow, which grows with both a decrease in cache size and increase in data set size. As described in Section 5.1, the data sets chosen for our applications are small in favor of reduced simulation time. The chosen ratio of data set to cache size in our measurements is conservative. Hence we expect the limitation of speculative state overflow to be even more severe in more realistic system configurations.

Figure 7 further shows that the implicit-only architecture is more sensitive to the cache configuration. The performance of Multiplex is nearly invariant of the cache size. This is a direct consequence of the fact that Multiplex incurs significantly less speculative state overflow than the implicit-only CMP.

5.6 Thread Size in Implicit-only CMPs

The implicit-only CMP exploits parallelism at the innermost loop level. This can be inefficient because of limited parallelism at this

Table 5: Speedup of eager vs. conservative explicit thread selection in Multiplex.

Benchmark	*apsi*	*applu*	*turb3d*	*tomcatv*
Eager explicit threading	1.33	1.30	1.62	3.37
Conservative explicit threading	1.73	1.97	1.94	2.34

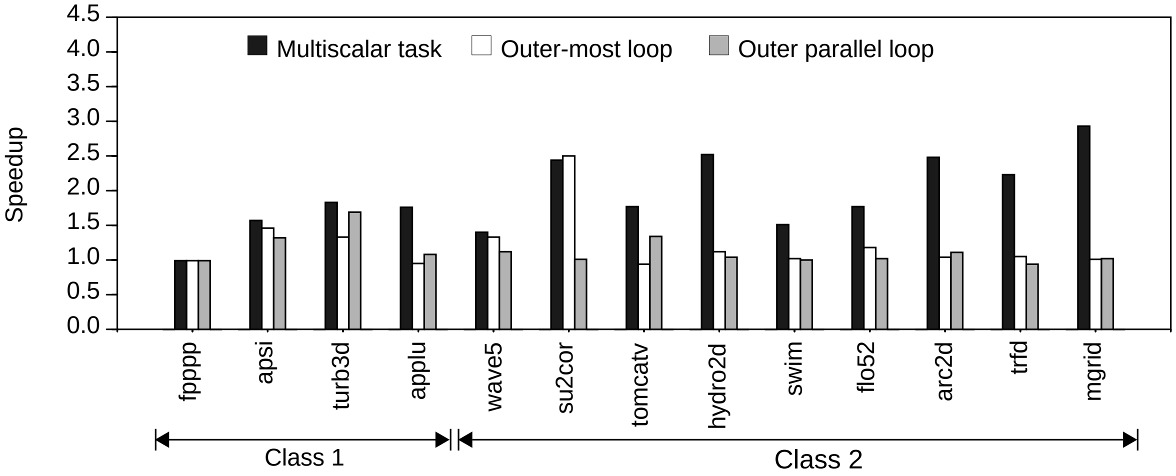

FIGURE 8: Impact of thread size in the Implicit-only CMP. The left bars indicate our base implicit-only CMP performance (from Figure 5). The right bars show the performance when selecting threads from outer parallel loops, as done for explicit threads in Multiplex. Note, that the presence of such outer loops depends on the compiler's ability to identify them. For example, in *fpppp* "inner" and "outer" loops are the same. The results show that compiler-only solutions for improving the performance of implicit-only CMPs face hard limits.

level. We have argued that Multiplex can reduce this inefficiency because it exploits compiler-detected parallelism in outer loops, which encompass the inner, possibly serial program sections. Figure 8 demonstrates that it would not be a simple solution for the implicit-only CMP to exploit the same outer-loop parallelism. In this experiment, we force the compiler to generate implicit threads (for the implicit-only CMP) consisting of (1) the iterations of outermost loops and (2) the iterations of outer *parallel* loops. Exploiting outermost loops increases the window size to find opportunities for parallelism. Exploiting outer *parallel* loops reduces the number of dependences and allows us to directly compare with Multiplex, which exploits the same loop parallelism.

Figure 8 shows that the selection of large threads in an implicit-only architecture would lead to a drastic performance degradation in most applications. Figure 9 shows the detailed overheads. The primary reason for the degradation is that the threads become so large that speculative state overflow dominates. For example, in *su2cor,* the overflow increases from 0 to 70% of the total number of cycles in the original, implicit-only execution. In *mgrid*, this overhead increases to 200%. The figure further shows that selecting threads from iterations of outermost loops can introduce significant overhead due to dependences. This overhead can be reduced when generating threads from outer *parallel* loops, as shown by the third bars in the figure. In order to generate dependence-free threads from parallel loop iterations, several state-of-the-art compiler techniques have been added to the code generator. The effect of selecting threads from outer parallel loops is two-fold. As expected, dependence-related overheads are significantly reduced. By contrast, speculative state overflow and load imbalance increases. We found that the increase in load imbalance is an indirect effect of the overflow, so is the increase of memory stalls in *trfd*. As Figure 6 shows, these overflow-related overheads drastically decrease when executing the same codes on Multiplex. Hence we can conclude that speculative state overflow is the most significant intrinsic, threading-related overhead of an implicit-only architecture and Multiplex can eliminate or significantly reduce this limitation.

6 RELATED WORK

There are several projects exploring architectural proposals for implicit threading such as Wisconsin Multiscalar [27,12] and Trace Processor [26], Stanford Hydra [15], CMU Stampede [28], Minnesota Superthreaded processor [31], Illinois Speculative NUMA [8], Speculative Multithreaded architecture [19], and SUN Microsystems MAJC [30]. While Multiplex proposes techniques to unify implicit and explicit threading within a single application, these projects have focused on employing implicit and explicit threading separately on a per application basis but not combined within one application.

Many of the projects have a compiler component to develop compiler techniques for implicit threading. Some of the projects use the SUIF compiler [14] for program analysis but rely on manual identification of program sections for speculative parallelization by the compiler [29,16]. Because misspeculation recovery is in software, the compiler also generates recovery code. While many of the projects evaluate performance on parts of applications selected for implicit threading [15,28,8], Multiplex evaluates entire applications by using an automated compiler infrastructure consisting of the Polaris compiler [4] integrated with the Multiscalar compiler [33]. Because speculative state buildup and misspeculation recovery is fully implemented in hardware, the Multiplex compiler does not generate any misspeculation recovery code.

In [22], the authors describe several compiler techniques to help thread-level speculation and argue that exploiting loop-level parallelism is insufficient. In [31], the authors describe compiler techniques for superthreaded architectures. No implementation of these techniques exist yet.

There are proposals to provide hardware support to make dependence tracking efficient in DSM systems. Extensions to compiler techniques for runtime data dependence testing and software misspeculation recovery are proposed in [35,34]. While these extensions focus on the specific compiler technique of runtime data-dependence testing, the Multiplex compiler performs general unification of implicit and explicit threads.

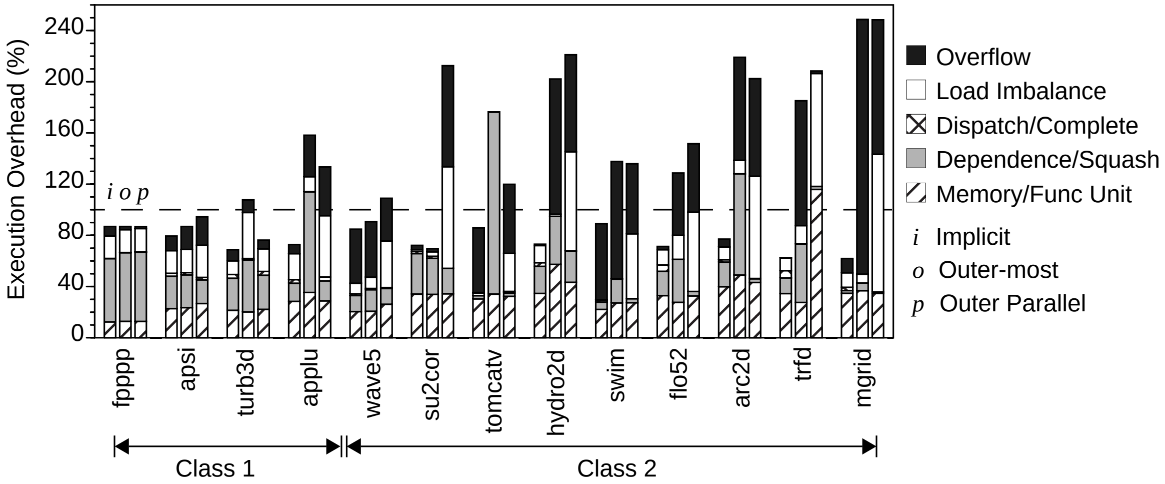

FIGURE 9: Overhead of different thread size in the implicit-only CMP. The figure depicts the overhead breakdown for the measurements from Figure 8. 100% represents the number of cycles in implicit-only execution.

7 CONCLUSIONS

Recent proposals for CMPs advocate speculative, or *implicit*, threading in which the hardware employs prediction to peel off instruction sequences (i.e., *implicit* threads) from the sequential execution stream and speculatively executes them in parallel on multiple cores. These proposals extend a conventional shared-memory multiprocessor, which employs *explicit* threading, with the ability to handle implicit threads. The proposals extend hardware cache-coherence protocols to handle speculation.

While the proposals focus on only implicitly-threaded code sections, this paper identified, for the first time, the issues in combining explicit and implicit threading to allow a single program to switch back and forth between implicit and explicit threads. We proposed the Multiplex architecture to unify implicit and explicit threading. We made the observation that the coherence protocol states required for explicit threading can be mapped to a subset of the protocol states required for implicit threading. We extended the implicit protocol by one extra state transition so that the entire explicit protocol can be emulated as a subset of the extended implicit protocol. This extension allows unifying the architecture for the two threading models without additional hardware.

Using a fully-integrated compiler infrastructure for automatic generation of Multiplex code, this paper presented a detailed performance analysis for entire benchmarks, instead of just implicitly-threaded sections, as done in previous papers. For the ten SPECfp95 and three Perfect benchmarks, we showed that neither an implicit-only nor explicit-only architecture performs consistently better than the other across the benchmarks. A CMP with four dual-issue CPUs achieves a speedup of 1.48 and 2.17 over one dual-issue CPU, using implicit-only and explicit-only threading, respectively. We showed that Multiplex matches or outperforms the better of the two architectures for every benchmark, and a four-CPU Multiplex achieves a speedup of 2.63.

Our detailed analysis indicated that the dominant overheads in an implicitly-threaded CMP are speculation state overflow due to limited L1 cache capacity, and load imbalance and data dependences in fine-grain threads. We also presented evidence that a naive thread selection (by the compiler) of explicit threads whenever possible would lead to inferior performance due to high explicit thread dispatch overhead in fine-grain threads.

8 ACKNOWLEDGMENTS

This work was supported in part by NSF grants #9974976-EIA and #9986020-EIA.

9 REFERENCES

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, Feb. 1993.

[3] M. Berry, D. Chen, P. Koss, D. Kuck, L. Pointer, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The Perfect Club Benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[4] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, pages 78–82, Dec. 1996.

[5] W. Blume and R. Eigenmann. Non-linear and symbolic data dependence testing. *IEEE Transactions on Parallel and Distributed Systems*, 9(12):1180–1194, Dec. 1998.

[6] S. Breach, T. Vijaykumar, and G. Sohi. The anatomy of the register file in a multiscalar processor. In *Proceedings of the 27th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 27)*, pages 181–190, Nov. 1994.

[7] B. Case. Spec95 retires spec92. *Microprocessor Report*, August 21 1995.

[8] M. Cintra, J. F. Martinez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 13–24, June 2000.

[9] F. Dahlgren. Boosting the performance of hybrid snooping cache protocols. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 60–69, 1995.

[10] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 58–67, May 1992.

[11] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 15–29, June 1991.

[12] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. Sohi. Speculative versioning cache. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, pages 195–205, February 1998.

[13] J. Gu, Z. Li, and G. Lee. Experience with efficient array data flow analysis for array privatization. In *Sixth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 157 – 167, June 1997.

[14] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.

[15] L. Hammond, M. Willey, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9), September 1997.

[16] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, October 1998.

[17] J. Hennessy. The future of systems research. *IEEE Computer*, 32(8):27–33, Aug. 1999.

[18] M. Horowitz, R. Ho, and K. Mai. The future of wires. In *Proceedings of the Semiconductor Research Corporation Workshop on Interconnects for Systems on a Chip*, May 1999.

[19] P. Marcuello, A. Gonzalez, and J. Tubella. Speculative multithreaded processors. In *Proceedings of the 1998 International Conference on Supercomputing*, 1998.

[20] A. Moshovos, S. E. Breach, and T. N. Vijaykumar. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[21] B. A. Nayfeh and K. Olukotun. Exploring the design space for a shared-cache multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 166–175, April 1994.

[22] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the Seventh International Conference on Parallel Architectures and Compilation Techniques*, Oct. 1999.

[23] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[24] B. Pottenger and R. Eigenmann. Idiom recognition in the Polaris parallelizing compiler. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 444–448, July 1995.

[25] W. Pugh. Going beyond integer programming with the omega test. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):204–211, Feb. 1995.

[26] J. E. Smith and S. Vajapeyam. Trace processors: Moving to fourth-generation microarchitectures. *IEEE Computer*, 30(9):68–74, Sept. 1997.

[27] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[28] J. G. Steffan, C. B. Colohan, A. Zhaia, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12, June 2000.

[29] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, pages 2–13, February 1998.

[30] M. Tremblay. An architecture for the new millennium. In *Proceedings of the 1999 Hot Chips Symposium*, August 1999.

[31] J.-Y. Tsai, J. Huang, C. Amlo, D. Lilja, and P.-C. Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 98(9), Sept. 1999.

[32] P. Tu and D. Padua. Automatic array privatization. In *Proceedings of the Sixth Languages and Compilers for Parallel Computing*, pages 500–521. Springer-Verlag, 1994.

[33] T. N. Vijaykumar and G. S. Sohi. Task selection for a multiscalar processor. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 31)*, December 1998.

[34] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, Jan. 1998.

[35] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for speculative parallelization of partially-parallel loops in dsm multiprocessors. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, Jan. 1999.