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

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

School of Electrical and Computer Engineering
Purdue University
1285 EE Building
West Lafayette, IN 47907
mux@ecn.purdue.edu, http://www.ece.purdue.edu/~mux

Abstract

Traditional monolithic superscalar architectures, which extract instruction-level parallelism (ILP) to achieve high performance, are not only becoming less effective in improving the clock speed and ILP but also worsening in design complexity and reliability across generations. Chip multiprocessors (CMPs), which exploit thread-level parallelism (TLP), are emerging as an alternative. In one form of TLP, the compiler/programmer extracts truly independent *explicit* threads from the program, and in another, the compiler/hardware partitions the program into speculatively independent *implicit* threads. However, explicit threading is hard to program manually and, if automated, is limited in performance due to serialization of unanalyzable program segments. Implicit threading, on the other hand, requires buffering of program state to handle misspeculations, and is limited in performance due to buffer overflow in large threads and dependences in small threads.

We propose the Multiplex architecture to unify implicit and explicit threading by exploiting the similarities between the two schemes. Multiplex employs implicit threading to alleviate serialization in unanalyzable program segments, and explicit threading to remove buffering requirements and eliminate small threads in analyzable segments. We present hardware and compiler mechanisms for selection, dispatch, and data communication to unify explicit and implicit threads within a single application. We describe the Multiplex Unified Coherence and Speculative versioning (MUCS) protocol which provides unified support for coherence in explicit threads and speculative versioning in implicit threads of an application executing on multiple cores with private caches. On the ten SPECfp95 and three Perfect benchmarks, neither an implicitly-threaded nor explicitly-threaded architecture performs consistently better across the benchmarks, and for several benchmarks there is a large performance gap between the two architectures. Multiplex matches or outperforms the better of the two architectures for every benchmark and, on average, outperforms the better architecture by 16%.

1 Introduction

Improvements in CMOS fabrication processes continue to increase on-chip integration and transistor count to phenomenal levels. Traditional monolithic superscalars use the rising 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 [25, 21,1] and ILP but also worsening in design complexity [20] and reliability [2] 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 [29,18]. 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.

The most common form of TLP is *explicit* threading used in conventional shared-memory multiprocessors. In explicit threading, 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. Explicit threads either compute independently or share and communicate data through memory when necessary. Examples of CMPs using explicit-threading are IBM Power4 [11] and Compaq Piranha [4]. Explicit threading's key shortcoming, however, is that it requires a programmer or parallelizing compiler either to guarantee that threads can compute independently or to coordinate shared accesses among threads through synchronization. Unfortunately, parallel programming is a tedious and costly task only suitable for high-end systems. Similarly, while parallelizing compilers have succeeded in threading many large and important applications, automatic parallelization has been limited to programs and program segments with statically analyzable dependences. When the compiler fails to prove independence, the corresponding program segment is executed serially on a single core.

Alternatively, 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. To preserve program execution correctness, implicitly-threaded hardware identifies and satisfies all dependences among implicit threads. Examples of proposed architectures using implicit threading are Multiscalar [29] and Trace Processor [28], Hydra [18], Stampede [30], Superthreaded processor [33], Speculative NUMA [10], and MAJC [32].

To maintain program correctness, implicitly-threaded architectures rely on the 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 [29]. While speculative buffer overflow results in complete stalling or rollback of speculative threads and essentially serialization of execution, buffering is *only* necessary if there are true dependences among threads that are not detectable at compile time. Implicit threading's key shortcoming is that the hardware must *always* buffer program state to track dependences among threads. State-of-the-art buffering techniques (e.g., custom buffering [13] and cache-based buffering [15, 19, 30, 10]), however, can only provide fast buffering large enough to accommodate short-running implicit threads (e.g., up to a hundred instructions). Small threads limit the scope of extracted parallelism, increase the likelihood of inter-thread dependence, and reduce performance.

We propose the Multiplex architecture for CMPs to unify implicit and explicit threading based on two key observations: (1) Explicit threading's weakness of serializing unanalyzable program segments can be alleviated by implicit threading's speculative parallelization; implicit threading's performance loss due to speculative buffer overflows in large threads and dependences in short threads can be alleviated by large explicit threads' exemption from buffering requirements in analyzable program segments. (2) To achieve high performance, explicit and implicit threading employ cache coherence and speculative versioning [15,30,19,10], respectively, which are similar memory hierarchy mechanisms involving multiple private caches for efficient sharing of data. Multiplex exploits the similarities to allow efficient implementation without much extra hardware and combines the complementary strengths of implicit and explicit threading to alleviate the individual weaknesses of the two schemes.

The main contributions of this paper are:

- we present architectural (hardware and compiler) mechanisms for selection, dispatch, and data communication to unify explicit and implicit threads from a single application;

- we propose the Multiplex Unified Coherence and Speculative versioning (MUCS) protocol which provides unified support for coherence in explicit threads and speculative versioning in implicit threads of a single application executing on multiple cores with private caches;
- using simulation of the ten SPECfp95 and three Perfect benchmarks, we show that neither an implicitly-threaded nor explicitly-threaded architecture performs consistently better across the benchmarks, and for several benchmarks there is a large performance gap between the two architectures;
- we show that Multiplex matches or outperforms the better of the two architectures for every benchmark and, on average, outperforms the better architecture by 16%.

In the following section, we describe advantages and disadvantages of current explicit and implicit architectures, and motivated the need for a unified architecture. In Section 3, we introduce Multiplex. Section 4 characterizes the key factors impacting performance and presents a qualitative performance analysis of TLP architectures. Section 5 presents the simulation methodology and results. Section 6 presents a summary of related work. Finally, Section 7 concludes the paper.

2 Background: Execution Models for Explicit & Implicit Threading

In this section, we briefly describe compare and provide examples for thread execution and the required hardware support in explicitly-threaded and implicit-threaded architectures. At the highest level, the key similarity between these architectures is that both simultaneously execute multiple threads that communicate among one another. As such, the dominant fraction of hardware resources required by either architecture is common and includes execution resources (e.g., multiple CPU cores) and communication resources (e.g., coherent shared memory through L1 caches).

The key differences are how, in each architecture, the hardware detects when communication is necessary — i.e., inter-thread data dependences — and identifies subsequent threads to execute upon thread completion — i.e., the inter-thread control dependences. 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). Because, software obviates the need for hardware to track the dependences, hardware achieves high performance by providing fast mechanisms for data communication, and thread dispatch and execution.

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. The hardware validates all control flow and data dependence speculations by verifying against the sequential program order, and triggers rollbacks upon detecting possible misspeculations which may violate sequential semantics.

In the rest of this section, we present example executions on each of the two (threading) architectures. We point out the key performance problems with the architectures to illustrate that combining the two can both (1) achieve higher performance by exploiting one architecture's strengths to alleviate the other architecture's weaknesses, and (2) be implemented efficiently without much hardware overhead by exploiting the similarities between the two architectures.

```
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;
}
```

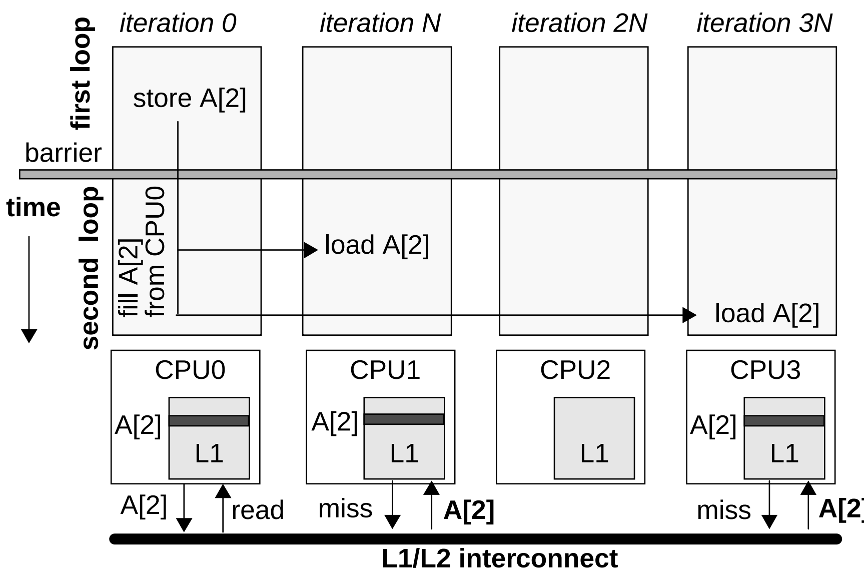

FIGURE 1: An example of explicit threading execution.

2.1 Example Execution in an Explicitly-Threaded CMP

Figure 1 shows a simple example of a program running on an explicitly-threaded CMP. The figure shows the high-level anatomy of a typical CMP with thread execution resources (i.e., four CPUs) and the data communication hardware (i.e., coherent L1 data caches). 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 implement 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 the example shown, the compiler (or the programmer) detects that the first loop and second loop are dependent only through array *A* and therefore separates them by a barrier synchronization. By identifying the *only* data dependence among the threads to be through *A[2]*, and specifying the dependence through the barrier primitive, software *guarantees* that hardware can otherwise execute the threads at peak speeds. Moreover, the "fork" primitive directs the hardware to execute exactly a single copy of *parallel_compute* as a thread on each CPU, specifying the thread control flow dependence. Unfortunately, when data dependences are unknown, the compiler (or the programmer) fails to generate explicit threads, and therefore executes the entire program segment (i.e., entire pair of loops) in a single thread.

2.2 Example Execution in an Implicitly-Threaded CMP

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 [35]) partitions the loop and assigns each implicit thread a single loop iteration. To help the hardware identify which subsequent threads to dispatch on the CPUs, each implicit thread includes (embedded in the executable) a list of *possible* subsequent threads, or *target* threads, and their starting program counters (not shown); the target threads are

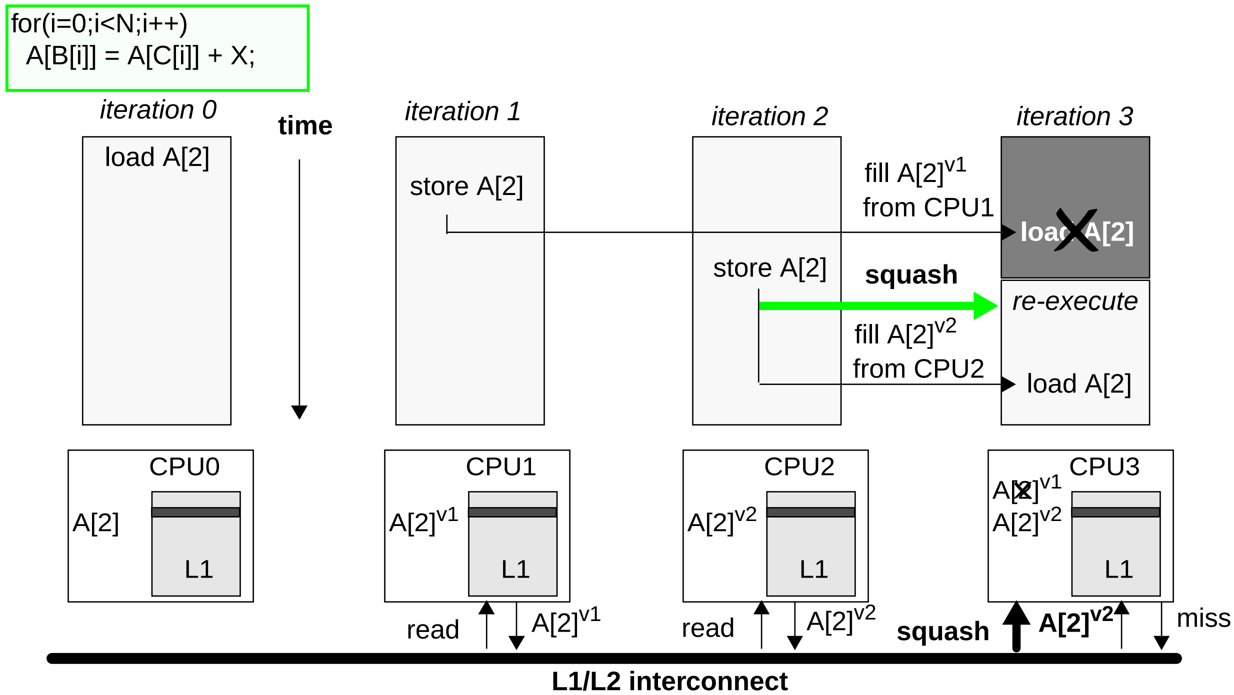

FIGURE 2: An example of implicit threading execution.

the exit points of a thread in the control flow graph. In this example, the execution of a loop iteration can either be followed by another loop iteration or the code following the loop (upon loop termination).

Unlike explicitly-threaded architectures, implicitly-threaded architectures rely on hardware prediction to dispatch threads. For every dispatched thread, hardware predicts a and selects among the thread's list of target threads, a subsequent thread to dispatch. 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. The loop branch condition at the end of each iteration verifies prediction for the subsequent iteration.

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. A key shortcoming of hardware data speculation is that 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 [15].

Unfortunately, implicitly-threaded architectures *always* predict and execute threads speculatively, and track data dependence in hardware even if a program segment is analyzable. For instance, in the example because there are no control flow dependences (e.g., a conditional break statement within the loop) except for the loop branch condition between the loop iterations and the code immediately following the loop, software can direct thread dispatch using a fork primitive, and obviate the need for hardware prediction and eliminating any potential misprediction overhead. Similarly, there are many scenarios where an advanced parallelizing compiler can either detect and guarantee no data dependences among threads exist [7,27,14] or can eliminate the data dependences (e.g., through array privatization [34,16]). In such scenarios, the hardware *unnecessarily* tracks data dependences, limiting the scope of parallelism to the buffering capacity in the L1 caches.

3 Multiplex: Unifying Explicit/Implicit TLP on a CMP

In this paper, we propose *Multiplex*, an architecture that unifies explicit and implicit threading on a chip multiprocessor. Multiplex alleviates explicit threading's weakness of serializing unanalyzable program segments by using implicit threading's speculative parallelization. Multiplex avoids implicit threading's performance loss due to speculative buffer overflows in large threads and dependences in short threads by using large explicit threads which are exempt from buffering requirements in analyzable program segments. Thus, Multiplex combines the complementary strengths of implicit and explicit threading to alleviate the individual weaknesses of the two schemes. Multiplex achieves efficient implementation without much extra hardware by exploiting the similarities between explicit threading's cache coherence and implicit threading's speculative versioning mechanisms.

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 from the program to execute on a CPU, (3) *data communication*, 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 selection, dispatch, and data communication to unify explicit and implicit threads within a single application.

Figure 3 illustrates a Multiplex CMP. Our Multiplex CMP is loosely derived from the Wisconsin Multiscalar [29,15]. As in traditional small-scale multiprocessors, Multiplex CMP includes a small number of conventional superscalar CPU cores with first-level instruction and data caches and a shared level-two cache [23]. To support implicit and hybrid explicit/implicit threading, Multiplex also 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

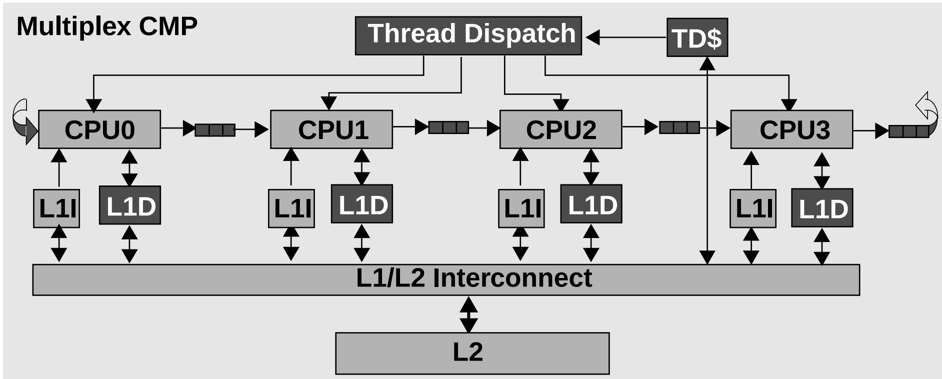

FIGURE 3: A Multiplex chip multiprocessor. The figure depicts the anatomy of a Multiplex chip multiprocessor. The blocks appearing in a light shade of gray are components used in a conventional (explicitly-threaded) multiprocessor architecture including the processing units, the L1 instruction caches, 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 level-one data caches, and the register communication mechanism.

compiler has the opportunity to choose between implicit and explicit 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.

Selecting explicit threads. Multiplex executes program segments which the compiler can partition into independent threads or threads with known data dependences as explicit threads. The compiler coordinates the known data dependences and sharing in such threads using explicit synchronization statements. Such threads maximize the parallelism exploited, eliminate hardware 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 [6], SUIF [17]) use a myriad of techniques to test [7,27,14] and eliminate data dependence in program segments [3,34,26,16]. Moreover, these compilers increase thread performance in analyzable program segments through code transformations to optimize for memory hierarchy locality and communication latency [17].

In explicit threading, thread size plays a key role in minimizing thread execution overhead. Dispatching explicit threads (Section 3.2) requires at a minimum setting up private stacks, passing arguments through the stacks, and synchronizing the threads upon completion. Explicit threading compilers typically partition the work among coarse-grain threads to amortize the dispatch and completion overhead over thread execution time. For instance, in nested loops with small inner loop bodies, explicit threads often consist of outer loop iterations [17,6]. Too coarse-grain a thread, however, increases the likelihood of load imbalance diminishing the opportunity for parallel execution.

When selecting explicit threads, the compiler has full flexibility in choosing how to partition the work among threads. In partitioning the code into explicit threads, the compiler can analyze and estimate the appropriate thread size based on the dispatch overhead and load imbalance. The compiler also has the flexibility of choosing the order in which explicit threads are dispatched (Section 3.2). Together, selecting thread size and dispatch order can help minimize load imbalance and dispatch overhead.

Selecting implicit threads. Multiplex executes program segments with control flow or data dependences that are unanalyzable at compile time as implicit threads 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 [35]. 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 a state-of-the-art compiler (e.g., the Multiscalar compiler [35]) to generate implicit threads. Alternatively, hardware rather than the compiler can extract and select implicit threads [28,12]. Selecting threads in hardware allows extracting TLP directly from uniprocessor application binaries at runtime, obviating the need to recompile the program. By selecting implicit threads in the compiler, Multiplex benefits from many key transformation techniques available at compile time to improve implicit thread performance [35].

There are two key criteria for implicit thread selection to minimize: (1) control-flow and data prediction and speculation overhead, and (2) data dependences and their distance among threads. An implicit thread typically includes one of or more (statically) adjacent basic blocks. Minimizing control-flow speculation overhead simply requires that the compiler carefully selects thread boundaries so that threads end at branch instructions with predictable outcomes (e.g., loop branches). This way, hardware prediction successfully dispatches threads and reduces speculation overhead. The Multiplex compiler exploits a number of techniques to analyze and reduce the overhead due to speculation and data dependence [35].

As in explicit threads, thread size plays a key role in implicit thread performance. Larger threads may help amortize the thread dispatch and completion overhead, and increase the scope for parallelism by reducing dependence among threads. However, data speculation overhead significantly limits thread size in implicit threading. Hardware must maintain all memory modifications by a speculatively executing implicit thread so that subsequent speculative threads can consume the results of past computation. Moreover, in case of a misprediction, all memory must be restored to a state conforming to sequential execution semantics. Therefore, there may be multiple versions of a data memory block present in processor caches, significantly increasing the memory overhead and cache traffic [35]. Consequently, implicit threading typically resorts to fine-grain (rather than coarse-grain) threads (e.g., inner loops) to exploit parallelism. Moreover, the speculation overhead constraint on thread size also limits the compiler's flexibility in varying thread size to reduce load imbalance.

Unifying thread selection in Multiplex. To minimize execution overhead, the Multiplex compiler always searches first for statically parallelizable program segments and partitions them into explicit threads. The compiler subsequently generates the rest of the program segments as implicit threads.

There are scenarios in which there is a trade-off between 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 compilers flexibility in choosing the threading model helps complement the strengths of both models, thereby improving application performance. In Section 5, we will present simulation results indicating how simple compiler heuristics help unify thread selection in Multiplex.

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 only requires copy semantics for stacks and registers (and not memory) upon dispatch.

Dispatching explicit threads. As in conventional explicitly-threaded architectures, Multiplex uses an application programming interface to dispatch threads. To minimize dispatch overhead, Multiplex supports the programming interface directly at the instruction set architecture level. 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.

In explicit threading, each thread uses a private stack. In Multiplex, the middleware (i.e., the system initialization library) is responsible to allocate private stacks for all CPUs. A `setsp` instruction assigns the pre-allocated stacks to individual CPUs. The `setsp` instruction takes two arguments in architectural registers. The first argument specifies the starting address of a private stack pointer, and the second argument specifies the which CPU's stack pointer is being set. The middleware need only to allocate private stacks once per application execution, and only re-allocate when a thread requires growing the stack.

Upon dispatch, explicit threading requires implementing a copy semantics in which the software (i.e., generated by the compiler or the programmer) passes data from the main (i.e., forking) thread's registers and stack to the dispatched threads. Compilers/programmers often encapsulate explicit threads into procedure bodies. As such the copy semantics for the threads is simply the incoming arguments into the procedure. In Multiplex, the middleware copies all the procedure arguments into the private stacks in the main thread prior to dispatch. The `fork` instruction dispatches a "wrapper" procedure on every CPU that reads the arguments off of the stack and passes them in the appropriate architectural registers. While copying arguments can be accelerated using hardware, explicit threads are often large enough that the software copying overhead becomes a small fraction of overall execution time.

Dispatching implicit threads. Multiplex dispatches implicit threads sequentially in program order [29]. A thread dispatch unit (Figure 3) uses the current implicit thread to predict and dispatch a subsequent thread. The compiler/programmer generates a thread descriptor and embeds it immediately prior to the thread code. The thread descriptor 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 [29]. To accelerate thread dispatch, a thread descriptor cache (Figure 3) caches recently referenced thread descriptors.

A novel aspect of implicitly-threaded architectures is on-demand data communication and renaming. In implicitly-threaded architectures, all necessary register and memory values produced by one thread and subsequently consumed by another are directly communicated through hardware on demand. As such, thread dispatching does not require any "copy" semantics. Moreover, all registers and memory addresses assigned to by one thread are renamed in hardware on demand. Therefore, stacks are automatically privatized for implicit threads; i.e., assigning a value to a stack address creates a distinct version of the corresponding memory location for the assigning thread. As such, implicit threads do not require private stacks.

Unifying thread dispatch in Multiplex. The ability to execute both explicit and implicit threads enables a Multiplex CMP to exploit both types of TLP within an application. Software, however, must inform the hardware which type of threading is used for a given program segment so that hardware can provide the appropriate execution support. Multiplex uses the thread descriptor to specify the threading type, and the hardware modifies the *mode bit* based on the specification. The mode bit is set for all the implicit threads. The mode bit is clear for the "wrapper" procedure used to dispatch explicit threads. Upon switching to explicit threading, the thread dispatch hardware unit stops fetching descriptors and dispatching threads.

3.3 Data Communication

Much like all modern architectures, Multiplex uses registers and memory to store program state. While implicit threads share both register and memory state among each other, similar to the Multiscalar architecture, explicit threads share only memory state and not register state, similar to conventional shared-memory multiprocessors. Accordingly, implicit threads communicate both register and memory values among each other and explicit threads communicate only memory values. Multiplex uses Multiscalar's register communication mechanism for register dependencies among implicit threads, and we do not discuss the details of the register communication mechanism and refer the reader to [8,29]. In this section we focus on memory data communication among both implicit and explicit threads.

In both explicit and implicit modes, 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 in both modes is to track the copies so that the correct copy is delivered on every memory access. In explicit mode, the memory system locates the correct copy for loads either from main memory if there is no cached dirty copy or from another cache if it has a dirty copy and for stores ensures that no stale copies exist in other caches (e.g., via invalidates). In implicit mode, the memory system provides similar support but in the presence of speculative loads and stores. For implicit loads, the memory system not only locates the correct version much like explicit loads, but also enforces store-to-load program order; the memory system tracks speculative loads to detect (and squash) any load that prematurely accesses a location before a previous store in program order is complete. For implicit stores, the implicit memory system creates a new (speculative) version for every (speculative) store and tracks the program order among the multiple speculative versions.

The key to maintaining correctness in both modes is the Multiplex Unified Coherence and Speculative versioning (MUCS) protocol which tracks the copies and versions of every cache block present in the system. In explicit mode, MUCS tracks the location of copies in the system and takes appropriate action on loads and stores. In implicit mode, MUCS tracks both the location and the program order among the versions.

From the standpoint of the memory system, the key similarity between explicit and implicit modes is that both cases track a cache block's multiple instances (copies and versions) via the protocol state. On a load or store access to a block, the access proceeds if the state of the block permits the access, and otherwise the access is deemed a miss, goes to the next level similar to a regular miss (i.e., tag mismatch), and the protocol locates the correct instance of the block. Both modes allow the common case of hits in the private caches to proceed at cache hit speeds without any protocol action, and only cache misses invoke protocol action involving some "global" protocol state checking which may be slow.

The key differences between the two modes are that (1) implicit mode requires the memory system to track loads and stores to enforce store-to-load order by squashing any prematurely executed loads, (2) while explicit mode allows multiple copies of only one version, implicit allows multiple versions to co-exist, and (3) implicit mode requires the memory system to differentiate between speculatively and non-speculatively accessed data and "commit" speculatively accessed data to non-speculative state if speculation succeeds. These differences, however, do not imply any major incompatibilities between the two modes in the overall handling of memory accesses, but rather that certain combinations of access types (i.e., loads or stores) and protocol states may require different protocol action. In particular, the common case of cache hits are as fast in implicit mode as they are in explicit mode, implying that the two threading schemes can be unified efficiently.

In the remainder of this section, we describe the details of the MUCS protocol and explain the unification of explicit and implicit modes in MUCS. Apart from the protocol state of the accessed block and the access type (i.e., load or store), MUCS uses the mode bit to know if the access is from an implicit or an explicit thread to take appropriate action.

Data Communication in explicit mode. As in conventional explicitly-threaded architectures, loads and stores that hit (i.e., find the block in a permissible state) in the L1 caches proceed without any protocol action. On a load miss, the bus snoops on the other caches and the cache with a dirty copy supplies the block and also updates main memory (much like the Illinois coherence protocol). If no dirty copy is found, the next level supplies the block. If the requesting cache is the only L1 cache holding the block, the block is

State bit	Action
use	set per access by implicit 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 altogether at implicit thread commit and squash
dirty	set by all stores in both modes; used to writeback a version on invalidation in explicit mode and version consolidation in both modes; cleared on writeback to next level in both modes
commit	set for the entire cache altogether at implicit thread commit and set per access in explicit thread; used in both modes to allow replacements of committed dirty versions; cleared on every implicit speculative access
stale	set only in implicit 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 altogether at implicit thread squash; used in both modes to force misses (if commit bit clear and squash bit set) on the next access to the block (commit and squash never both set); cleared on every implicit access, 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 explicit invalidation, and in both modes for all committed/squashed versions other than the consolidated version

Table 1: MUCS protocol state and actions.

marked exclusive to optimize for future writes. On a store miss, the requesting cache obtains the block in the same manner as a load miss, but additionally, all other cached copies are invalidated.

There is one minor difference between conventional coherence protocol actions and MUCS' actions for explicit mode accesses. Because accesses in implicit mode need to differentiate between speculatively-accessed and committed blocks, MUCS uses some state bits for this purpose. As such, explicit mode accesses (loads and stores) are not speculative and do not require any enforcement of store-to-load order. Therefore, MUCS simply marks the blocks accessed in explicit mode as committed using the same state bits, making explicit mode accesses indistinguishable from committed implicit mode accesses, and unifying accesses from both implicit and explicit modes within the same protocol.

Data Communication in implicit mode. Because implicit mode loads are speculative, MUCS sets the *use bit* to record speculative loads. The key purpose of marking speculative loads is if a preceding CPU (i.e., in cyclic thread dispatch order) performs a store to the same block after the load, the store's invalidation triggers a squash of the premature load. Load misses issue a bus request and MUCS supplies a copy of the closest preceding CPU's (dirty speculative) version to the requesting cache; if the preceding CPUs do not have a dirty version, then the next level supplies the block. When the thread commits, all the use bits in the entire cache are cleared altogether. Although, blocks not accessed by the thread also have their use bits cleared redundantly, this global clearing avoids scanning the cache for the blocks touched by the committing thread.

MUCS uses the usual *dirty bit* to record stores. If a thread loads before storing to the same block, both of the use bit and dirty bit are set. Unlike conventional invalidation-based coherence protocols, stores do not always invalidate the other CPUs. On store misses, the bus snoops on the other caches and if there are any succeeding CPUs with the use bit set, that CPU (and all subsequent CPUs) is squashed. If any succeeding CPU holds the block but has not loaded or stored to it (i.e., the block exists from some previous implicit thread), then the block is marked as potentially stale (and not definitely stale because the store is speculative, and may be squashed later) with the *stale bit*, so that the current thread or any future thread executing on the CPU is forced to miss on the block and obtain the most recent version. On the preceding CPUs, the use bit is ignored, but the stale bit is set to force a miss for future threads that execute on the CPU. The preceding CPUs merely set the stale bit *without* invalidating the block and continue to use the block because the block contains valid data for the current thread, and the data is stale only for threads future to the storing thread.

Accesses (loads and stores) from speculative threads cannot be replaced because eviction of a speculative block would cause MUCS to lose track of possible violations of program order. Speculatively loaded blocks may be distinguished through the use bit, but speculatively stored blocks are inseparable from non-speculatively stored blocks because both have the dirty bit set. MUCS sets the *commit bit* on thread commits and clears the bit (i.e., commit bit cleared indicates speculative) on accesses (loads and stores) from speculative threads (i.e., all implicit threads except for the earliest thread which is non-speculative). When the thread commits, all the commit bits in the entire cache are set altogether. Much like the redundant clearing of use bits, this global setting avoids scanning the cache for the blocks touched by the committing thread.

The storing CPU creates a dirty, speculative version after obtaining a copy of the closest preceding CPU's version. Thus, multiple speculative versions of the same block co-exist in the system. Because the stale bit on a block indicates the potential existence of versions that are future to the block, any cache fill on a load or store miss sets the stale bit if the corresponding bus snoop detects uncommitted dirty versions in succeeding CPUs.

If a thread is squashed then the commit bits are not set. But commit bits alone cannot distinguish between blocks accessed in the squashed thread and blocks accessed in the middle of a yet uncommitted thread. To avoid scanning the cache for invalidating the blocks touched by a squashed thread, MUCS uses the *squash bit*, and sets all the squash bits in the entire cache altogether irrespective of whether the thread accessed a certain block or not. MUCS also clears all use bits in the entire cache altogether on a thread squash. Access to a block with the squash bit set and commit bit clear forces a miss and the squash bit is cleared when the miss is serviced. The squash bit is ignored if the commit bit is set because the squash bit is set globally even for the blocks not accessed by the squashed thread. Much like every (speculative) access clears the commit bit, every access clears the squash bit.

Although a block accessed by a squashed thread contains invalid data, the block's stale bit holds valuable information. Every store from a speculative thread marks all other versions as (potentially) stale, and even if the thread is squashed later the stale bits are not corrected immediately to avoid scanning the cache on squashes. Instead, the incorrectly marked stale bit on the most recent non-squashed version is corrected on the next access to the block. On the next access to the squashed block, the CPU misses and MUCS identifies the closest preceding committed version (which is incorrectly marked as stale) to the squashed version by following the reverse of the cyclic thread dispatch order, and clears the closest preceding version's stale bit.

On a miss, establishing the program order among multiple (committed and squashed) versions is central to maintaining correctness. MUCS uses the cyclic thread dispatch order among the CPUs to infer the program order among multiple versions. The currently speculative (i.e., not committed and not squashed) versions can be ordered using the current cyclic order among the CPUs, but ordering committed or squashed versions using the cyclic order fails because the position of the oldest thread in the system cyclically rotates from one CPU to the next (as threads commit). While a CPU may be older than another CPU at a certain point in execution, the cyclic rotation may result in the second CPU being older than the first in another point in execution. For example, the cyclic order at one point may be CPU3, CPU0, CPU1, and CPU2, and at that point CPU3 is older than CPU2, and then the order becomes CPU0, CPU1, CPU2, and CPU3, and now CPU2 is older than CPU3.

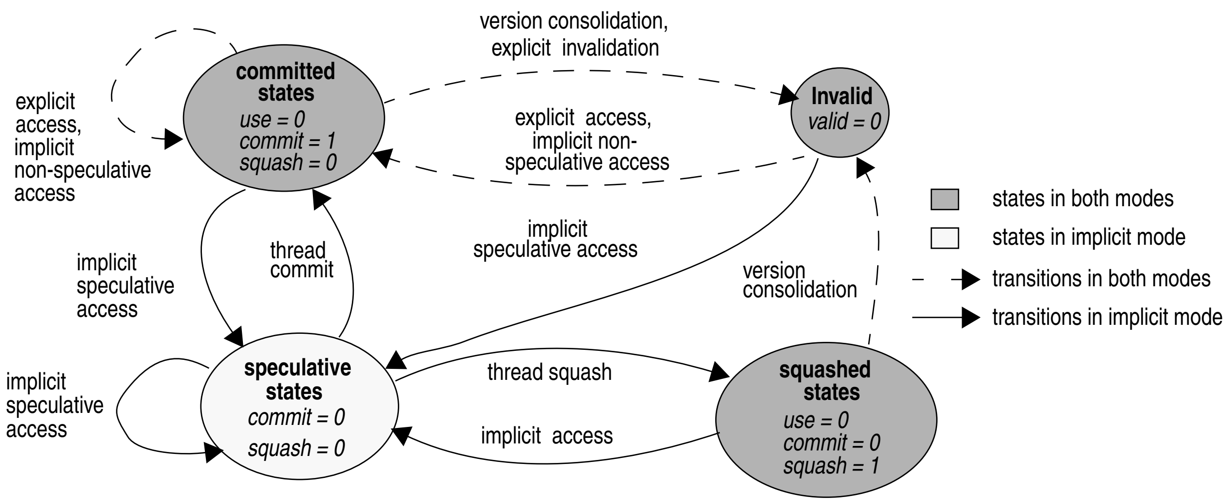

FIGURE 4: A high-level state transition diagram for MUCS.

This phenomenon is a problem only if versions from different cyclic orders are allowed to co-exist. To avoid this problem, MUCS *consolidates* the versions from a previous cyclic order at the next access. Versions from a previous cycle order are guaranteed to be committed or squashed and MUCS locates and writes back the most recent committed dirty version (i.e., the committed, non-stale, dirty version or the closest committed dirty version preceding the squashed, non-stale version) and invalidating all the other committed/squashed versions. Version consolidation needs to done only for the previous cyclic order's versions all of which are either committed or squashed, and does not change any of the speculative versions from the current cyclic order. Note that the write back at consolidation is no different than the writeback of a dirty block in conventional coherence protocols when either the block is invalidated due to another CPU's write request or a copy is made to satisfy another CPU's read request. Table 1 summarizes the protocol state bits.

Speculatively accessed (i.e., not committed and not squashed) block cannot be replaced. If a committed or squashed block needs to be replaced, then MUCS writes back the most recent committed version among the existing versions and all other committed/squashed versions are invalidated.

Switching between implicit and explicit modes. Because explicit threads may access data which was last accessed by an implicit thread, explicit threads may access blocks with squash or stale bit set. MUCS treats these explicit mode accesses as misses, much like these were implicit mode accesses, 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. Figure 4 shows a high-level state transition diagram of MUCS, illustrating the states and transitions shared by both modes and the states and transitions relevant to only implicit mode. There is no overhead for switching from one explicit thread to another explicit thread or from one implicit thread to another implicit thread, much like conventional multiprocessors and Multiscalar SVC. Except for the restrictions on replacement mentioned above, there is no scanning the cache, or no extra expensive cleanup at the beginning or end of implicit or explicit threads. MUCS performs all actions on demand, on a per access basis without maintaining any list of accessed blocks and triggering bus snoops on the blocks in the list.

Conventional coherence protocols employ optimizations such as using exclusive state and snarfing of data off the bus during a bus transaction by CPUs other than the sending and the receiving CPU involved in the transaction. These optimizations have been employed by Multiscalar SVC and are applicable to MUCS as well. Much like other speculative protocols [15,31,19,10], MUCS also can optimize away false squashes

Machine	Thread Size	Load Imbalance	Data Dependence	Thread Dispatch/ Completion Overhead	Data Speculation Overhead
Explicit-Only	Mostly coarse-grain threads; serial for unanalyzable segments	Huge for serial segments; high when threads are not multiples of CPUs	Low to none except for serial segments	Low for coarse-grain threads	None
Implicit-Only	Fine-grain threads	High when irregular control flow within thread	High	High	High for large threads
Multiplex	Coarse-grain explicit threads + fine-grain implicit threads	High *only* for implicit threads with high load imbalance	High *only* for implicit threads	High *only* for implicit threads	High *only* for large implicit threads

Table 2: Factors affecting performance in Explicit-Only, Implicit-Only, and Multiplex architectures.

by maintaining protocol state at word or smaller granularity instead of 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 Factors Affecting Performance

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. Table 2 depicts the factors affecting performance and summarizes their impact on explicit-only, implicit-only, and Multiplex CMPs. In the rest of this section, we briefly and qualitatively evaluate the impact of each factor on performance. 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 (as discussed in Section 3.1). Larger threads help (1) increase the scope of parallelism, and reduce the likelihood of data dependence across threads, and (2) reduce the impact of thread dispatch/completion overhead. Larger threads, however, increase speculation overhead 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 prevent 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.

Besides serial program segments, another factor contributing to load imbalance in explicit-only CMPs is the number of explicit threads when it is not evenly divisible by the number of CPUs; as a result, one or more CPUs idle until the completion of the program segment. Multiplex can partition the work so that the fraction resulting in a load imbalance in explicit-only CMPs executes as implicit threads, exploiting parallelism across program segments and eliminating the load imbalance.

In implicit-only CMPs, load imbalance is highly dependent on 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 (because the compiler knows exactly which threads produces and consume register values), memory dependences may incur additional speculation overhead when memory synchronization hardware is unable to prevent unwanted speculation [22]. 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, only use fine-grain threads when the compiler can not analyze dependences among larger thread bodies, e.g., when the compiler selects inner loops as threads because the outer loops are not analyzable. Multiplex can execute such fine-grain threads as implicit threads, thereby reducing the thread dispatch/completion overhead.

While thread dispatch incurs minimal overhead in implicit-only CMPs, thread completion incurs the overhead of flushing the load/store queue as in explicit-only CMPs, and may incur a high overhead. Thread completion overhead in implicit-only CMPs may be significant because these CMPs often use fine-grain threads. Multiplex reduces much of the thread completion overhead as compared in implicit-only CMPs by executing analyzable program segments in coarse-grain explicit threads.

Data speculation overhead. Data speculation in implicit-only CMPs is limited by the amount of buffering data caches can provide (Section 3.3). Speculation requires buffering all created 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 speculation overhead by execution independent threads as explicit threads, obviating the need for speculation.

5 Quantitative Performance Evaluation

In this section, we quantitatively evaluate a Multiplex CMP's performance using simulation. We first describe our compiler infrastructure, the experimental methodology, and the application and input parameters we use. In the base case performance results, we compare a Multiplex CMP with conventional explicit-only and implicit-only CMPs. Next, we present the results from two experiments providing evidence that (1) implicit-only CMPs are limited to using fine-grain threads and increasing thread size reduces performance in these machines, and (2) our heuristics-based compiler optimizations make a near-optimal decision in choosing between explicit and implicit threads.

5.1 Methodology and Infrastructure

We have developed a cycle-accurate simulator for a Multiplex CMP. Our simulator models multiple ILP CPU cores and pipelines, the memory hierarchy, and an implementation of the Multiplex threading mecha-

Processing Units	
CPUs	4 dual-issue, out-of-order
L1 i-cache	8K, 2-way 1 cycle hit
L1 d-cache	16K, 4-way, 16-byte block 1-cycle hit, byte-level disambiguation
Squash buffer size	64 entries
Reorder buffer size	64 entries
LSQ size	64 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
L2	9-cycle hit and transfer, perfect hit rate
L1/L2 interconnect	snoopy split-transaction bus, 32-bit wide

Table 3: System configuration parameters.

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

Table 4: Applications and input sets.
* indicates scaled down number of loop iterations in the interest of reduced simulation time.

nisms in detail. Table 3 summarizes the processor and system configuration parameters we use in this study. The CMP include four dual-issue out-of-order cores, each with L1 instruction and data caches. We assume perfect L2 hit rates, but model the cache fill latency between L1 and L2 and contention at the interconnect accurately.[1] 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 bus protocol.

Our compiler infrastructure integrates Polaris [6], a state-of-the-art parallelizing preprocessor generating explicit threads, with the Multiscalar compiler [35], a GCC-based compiler for generating implicit threads. Our compiler infrastructure is *fully* automated, obviating the need for hand tuning. Moreover, we compile the benchmarks as is without modifying the distributed 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 SPEC95 [9] and the Perfect [5] suites. Table 4 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. The change in input set, however, has a minimal impact on our performance results since the inherent communication/computation characteristics of the applications remain the same.

1. Our application data sets have small L2 footprints, and therefore our L2 assumption will have minimal impact on our performance results.

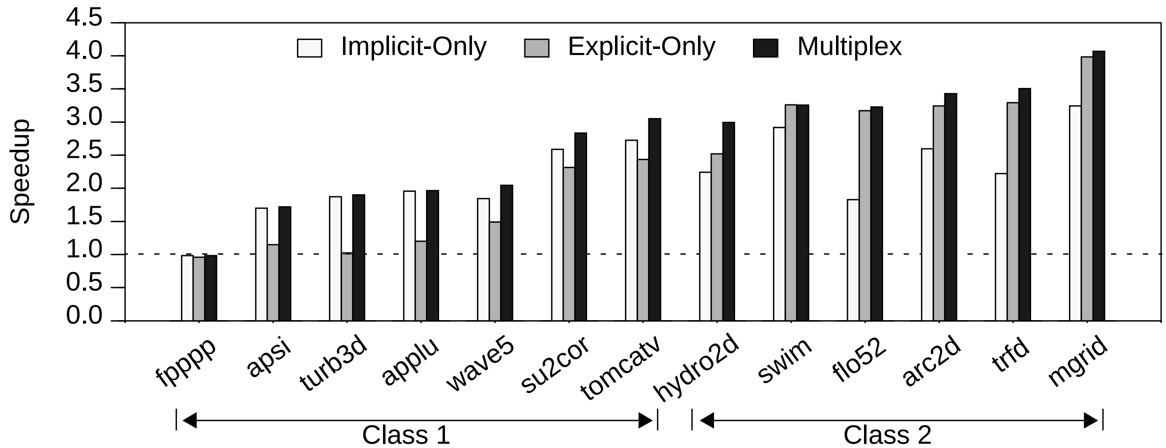

FIGURE 5: Overall Performance of Multiplex CMP compared to an implicit-only and an explicit-only CMPs. In class 1 applications, implicit-only outperforms explicit-only and vice versa in class 2 applications. In all applications, Multiplex matches or exceeds the performance of the better alterarnative.

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. 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 35% higher speedups and at best 85% higher speedups than the explicit-only CMP. In contrast, in class 2 applications the explicit-only CMP achieves on average 32% higher speedups and at best 74% 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 average achieves a speedup of 2.69, improving speedups by 16% over explicit-only and 22% 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 rest of this section.

Threading opportunity in the explicit-only CMP. Table 5 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 architectures, which can be overcome by Multiplex through executing the serial sections as implicit 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

Table 5: Fraction of the threaded execution time of each application that is recognized as parallel by the compiler and converted to explicit threads.

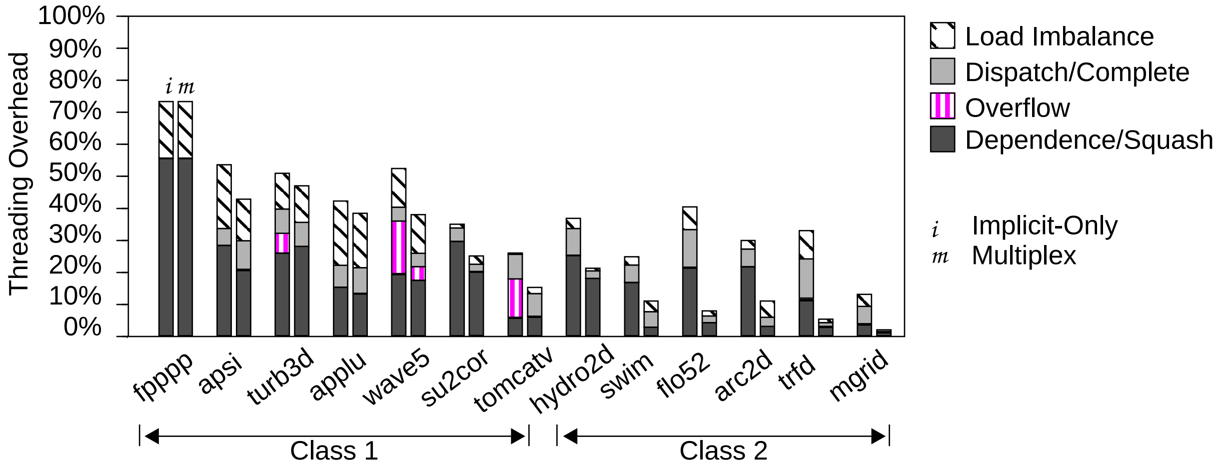

FIGURE 6: Overheads of the Implicit-only and the Multiplex architecture.

In addition, two of the class 1 applications, *apsi* and *applu* suffer from a small thread size. The outermost loops that the compiler can select as explicit threads in these applications consist of loops with small bodies and iteration counts. Therefore, the thread dispatch overhead in these applications significantly impacts overall thread execution time. Multiplex can select these loops as implicit threads, significantly improving performance over the explicit-only CMP.

Threading overhead in the implicit-only CMP. Figure 6 shows the overheads that have a first order impact on implicit-only and the Multiplex CMPs' performance. The figure plots overhead (i.e., the number of processor cycles not contributing to computation) as a fraction of overall execution time in the implicit-only CMP. The figure only includes overheads due to the threading mechanisms (discussed in Section 4); overheads intrinsic to the base superscalar CPU cores (e.g., pipeline hazards) are the same in all the CMPs and are not shown. For all applications, other system characteristics remain the same across systems, with the exception of memory latency in *mgrid*; using explicit threads to parallelize outermost loops in Multiplex changes *mgrid*'s data layout in the caches, significantly increasing data locality as compared to the implicit-only CMP.

We will first consider overheads in the implicit-only CMP and then discuss the changes when going to Multiplex. The figure shows that the largest 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 [22]) 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 [22].

Load imbalance is another significant factor especially in class 1 applications. *Fpppp*, *apsi*, *turb3d*, *applu*, and *wave5* have control flow irregularities in the inner loops (i.e., loop iterations including input-dependent conditionals [35]). Because the implicit-only CMP is limited to using fine-grain threads, it primarily targets inner loops and therefore suffers from load imbalance in these applications. Similarly, thread completion overhead of flushing the load/store queues is non-negligible in many of the applications due to the small thread size.

Finally, data speculation overhead due to speculative state overflow is only a significant overhead in *turb3d*, *wave5*, and *tomcatv*. For the implicit-only CMP, the compiler carefully selects thread size to minimize the state overflow [35]. In Section 5.3, however, we show that using larger threads to increase paral-

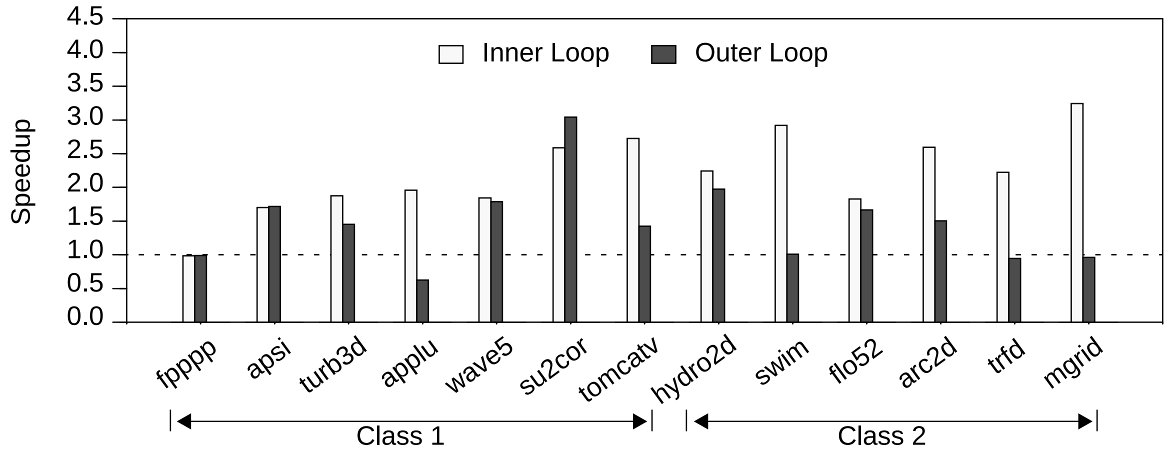

FIGURE 7: Effect of increasing the thread size in the Implicit-only CMP. The left bars show the same performance as in Figure 5. The right bars show the performance when selecting threads from outer parallel loops, as done for explicit threads in Multiplex. Note, that the presence of such outer loops depends on the compilers ability to identify them. For example, in *fpppp* "inner" and "outer" loop are the same.

lelism and eliminate dependences would prohibitively increase the speculation overhead. This overhead is one of the key limitations of implicit-only architectures and a motivation for Multiplex.

Threading overhead in Multiplex. The figure indicates that Multiplex reduces much of the overhead in the implicit-only CMP especially in class 2 applications. In these applications, Multiplex exploits advanced parallelization techniques to eliminate data dependences and generate coarse-grain explicit threads (consisting of outer loop nests), reducing all sources of overhead, and significantly improving performance over the implicit-only CMP.

In class 1 applications, there are a few program segments that Mutiplex converts to explicit threads. These explicit threads reduce the data dependence overhead in *apsi* and *su2cor*. Moreover, the explicit threads virtually eliminate the data speculation overhead in *tomcatv* and *turb3d*, and diminish it substantially in *wave5*. As such, *tomcatv* and *su2cor* exhibit a high performance boost from Multiplex. Unfortunately, the parallelization techniques the compiler uses slightly increase the instruction count, the overall impact on performance in *apsi* and *wave5* is modest.

5.3 Impact of Thread Size

A significant source of inefficiency of the implicit-only CMPs are synchronized serial regions. We have argued that Multiplex can reduce this inefficiency because it exploits explicit parallelism in outer loops, which encompass the inner serial program sections. Figure 7 demonstrates that it would not be a simple solution for the implicit-only CMP to exploit outer parallelism. In this experiment, we force the compiler to generate implicit threads (for the implicit-only CMP) consisting of the iterations of outer, parallel loops — the same loops selected as explicit threads in the explicit-only CMP (and Multiplex). Because these loops are dependence-free, the only source of overhead in the implicit-only CMP is due to data speculation and speculative state overflow.

The figure shows that this change would lead to a drastic performance degradation in most applications. The reason is that the threads become so large that speculative state overflow becomes dominant. For example, in *applu*, the overflow increases from 0 to 30% of the total number of cycles in the original, implicit-only execution. In *swim*, this overhead increases to 160%.

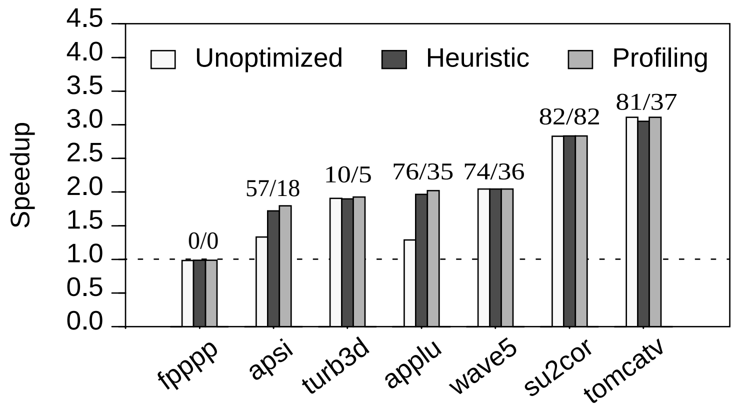

FIGURE 8: Reducing explicit thread dispatch overhead in class 1 applications. The figure illustrates the effect of the heuristic-based thread selection algorithm in eliminating high-overhead explicit threads in class 1 applications. Profiling shows optimal thread selection. Explicit thread dispatch overhead is minimal in class 2 applications, which benefit from coarse-grain threads. Numbers above the bars show the percentage of the execution time spent in explicit threads before and after applying the heuristic.

5.4 Reducing Dispatch Overhead in Explicit Threads

Small explicit threads can incur significant dispatch overheads for setting up and initializing private stacks. Because of this reason, small parallel loops run more efficiently using implicit threads. The compiler applies a simple heuristic to decide when it is better to run a compiler-recognized parallel loop with implicit rather than explicit threads. The heuristic "predicts" that all innermost loops run more efficiently with implicit threads. Inner loops are usually small and do not cause speculative state overflow in implicit threads. Hence there is usually no benefit from running such loop iterations as explicit threads.

Figure 8 shows the performance impact of this compiler optimization. It shows Multiplex's performance without and with applying the heuristic. In two applications, *apsi* and *applu*, there is a significant performance improvement. The numbers above the bars show the percentage of the execution time that is spent in explicit threads before and after applying the heuristic. In two applications (*fpppp and su2cor*) there is no significant change. In *fpppp*, there was no significant, compiler-recognized parallelism. In *su2cor*, the heuristic does not detect unprofitable explicit threads. In *tomcatv*, the heuristic does not improve performance because one of the threads incurs speculative state overflow after convertion to implicit. In *wave5* and *turb3d*, all affected threads perform identically before and after the optimization.

The figure also shows an upper performance bound that can be achieved by always correctly choosing the better of explicit and implicit threads. We obtained this bound by profiling the implicit-only and explicit-only executions and then manually combining the best cases loop by loop. The figure shows that the heuristic is already close to the optimum. Note that increasing the input data size, however, may lead to more speculative state overflow in inner loops and thus change the trade-off between implicit and explicit threads.

6 Related Work

There are many projects exploring architectural proposals for implicit threading such as Wisconsin Multiscalar [29,15] and Trace Processor [28], Stanford Hydra [18], CMU Stampede [30], Minnesota Superthreaded processor [33], Illinois Speculative NUMA [10], and SUN Microsystems MAJC [32]. 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 advanced SUIF compiler [17] for program analysis but rely on manual identification of program sections for speculative parallelization by the compiler [31,19]. 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 [18,30,10], Multiplex evaluates entire applications by using a fully automated compiler infrastructure consisting of the Polaris compiler [6] integrated with the Multiscalar compiler [35]. Because speculative state buildup and misspeculation recovery is fully implemented in hardware, the Multiplex compiler does not generate any misspeculation recovery code.

In [24], the authors describe several compiler techniques to help thread-level speculation and argue that exploiting loop-level parallelism is insufficient. In [33], the authors describe compiler techniques for super-threaded 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 [37,36]. 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.

7 Conclusions

Chip multiprocessors (CMPs), which exploit thread-level parallelism (TLP), are emerging as an alternative to traditional superscalar architectures. In one form of TLP, the compiler/programmer extracts truly independent *explicit* threads from the program, and in another, the compiler/hardware partitions the program into speculatively independent *implicit* threads. However, explicit threading is hard to program manually and, if automated, is limited in performance due to serialization of unanalyzable program segments. Implicit threading, on the other hand, requires buffering of program state to handle misspeculations, and is limited in performance due to buffer overflow in large threads and dependences in small threads.

We proposed the Multiplex architecture for CMPs to unify implicit and explicit threading based on two key observations: (1) Explicit threading's weakness of serializing unanalyzable program segments can be alleviated by implicit threading's speculative parallelization; implicit threading's performance loss due to speculative buffer overflows in large threads and dependences in short threads can be alleviated by large explicit threads' exemption from buffering requirements in analyzable program segments. (2) To achieve high performance, explicit and implicit threading employ cache coherence and speculative versioning, respectively, which are similar memory hierarchy mechanisms involving multiple private caches for efficient sharing of data. Multiplex exploits the similarities to allow efficient implementation without much extra hardware and combines the complementary strengths of implicit and explicit threading to alleviate the individual weaknesses of the two schemes.

We presented architectural (hardware and compiler) mechanisms for selection, dispatch, and data communication to unify explicit and implicit threads from a single application. We proposed the Multiplex Unified Coherence and Speculative versioning (MUCS) protocol which provides unified support for coherence in explicit threads and speculative versioning in implicit threads of a single application executing on multiple cores with private caches. Using simulation of the ten SPECfp95 and three Perfect benchmarks, we showed that neither an implicitly-threaded nor explicitly-threaded architecture performs consistently better than the other across the benchmarks, and for several benchmarks there is a large performance gap between the two architectures. We showed that Multiplex matches or outperforms the better of the two architectures for every benchmark and, on average, outperforms the better architecture by 16%.

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] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 32)*, Nov. 1999.

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

[4] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] K. Diefendorff. Power4 focuses on memory bandwidth. *Microprocessor Report*, 13(13), 1999.

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

[13] M. Franklin and G. S. Sohi. ARB: A hardware mechnism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

[21] M. Horowitz, R. Ho, and K. Mai. The future of wires. In *Proceedings of the Semiconductor Research Corporation Work-*

shop on Interconnects for Systems on a Chip, May 1999.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

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

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.