Architectural Considerations for Efficient Software Execution on Parallel Microprocessors

Srinivas Vadlamani and Stephen Jenks
Dept. of Electrical Engineering and Computer Science
University of California, Irvine, CA, USA
{srinivas.v, sjenks}@uci.edu

Abstract

Chip Multiprocessors (CMPs) and Simultaneous Multithreading (SMT) processors provide high performance but put more pressure on the memory interface than their single-thread counterparts. The "memory wall" problem is exacerbated by multiple threads sharing a memory interface, and will get worse as more cores are added. Therefore, communications between cores, using shared caches or fast interconnects between private caches, are needed to keep the CPUs busy without burdening the memory interface. Multiple CMP systems add another dimension to this challenging problem, as the communication mechanism is no longer uniform. To parallelize data-intensive applications for high performance on these systems, one must explore a number of execution behaviors in a complex architecture-dependent exercise that entails identifying key components of the communication subsystem and understanding their behavior under varying workloads. As part of ongoing research into efficient program execution models for parallel microprocessors, we have developed a tool to evaluate the performance of the storage controllers at different levels of the memory hierarchy under varying workloads and measure cache coherence overhead. The tool allows exploration of architectural features of real processors that affect the performance of several parallel execution approaches. Here, we demonstrate its use by evaluating two of our parallel programming models that employ architecture-specific optimizations and compare them to a conventional model for several applications on parallel microprocessors.

1 Introduction

With the introduction of parallel microprocessors, in the form of Simultaneous Multithreading (SMT) processors and Chip Multiprocessors (CMPs), the trend in commercial microprocessor design has shifted from exploiting instruction level parallelism (ILP) and ever increasing clock rates to exploiting thread level parallelism (TLP). SMT processors [20] maintain multiple executable thread contexts in hardware. The threads execute simultaneously and share the execution resources of the physical processor, such as the functional units and the caches. However, each thread has its own instance of the architectural state, which includes the program counter and registers. Chip Multiprocessors [13, 2] are built with several processor cores in a single package. They also share some levels of the memory hierarchy between the cores, but the amount of sharing varies by manufacturer and product. Cores tend to have their own individual first level caches. Some CMPs, such as Intel's Core Duo [13] series share large second level caches between cores, while others, like AMD's Athlon 64 X2 [2] have L2 caches that are private to each core.

The fast interconnects and/or one or more shared levels of the memory hierarchy in these parallel microprocessors present new opportunities for communication between the execution contexts/cores. But these resources could turn out to be potential bottlenecks if there are any hidden costs involved in using them, or if their performance is greatly reduced because of contention. Moreover, because of limited pin-out, or because some of these parallel microprocessors are pin compatible with their sequential predecessors, their memory interfaces tend not to scale with the number of threads. Often, the same memory interface that was already a bottleneck for a single-threaded CPU is used in the parallel microprocessors exacerbating the memory wall problem when multiple threads access memory simultaneously.

Executing parallel applications, particularly data-intensive ones, on these parallel microprocessors involves choosing an execution paradigm that takes into account the capabilities of the memory and cache controllers and the expense of maintaining cache coherence in the system. For example, we have shown that for important classes of applications, using the conventional parallel partitioning

1-4244-0910-1/07/$20.00

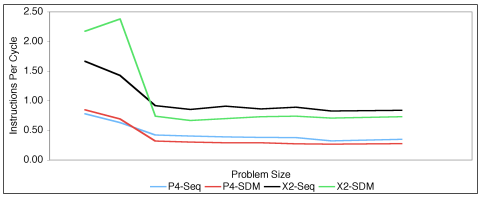

Figure 1. IPC measurements for FDTD

(the Spatial Decomposition Model, or SDM) tends to overwhelm the cache and memory interface on SMT processors, thus sometimes causing them to run slower than the sequential version of the programs [28]. This effect can also be seen in Figure 1, which compares the instructions per cycle (IPC) rate achieved by the Sequential (Seq) and SDM versions of an extremely memory-intensive Finite Difference Time Domain (FDTD) electromagnetic simulation running on a 3 GHz Intel Pentium 4 and a 2.4 GHz dual-core AMD Athlon X2. As long as the problem size fits in the 512K L2 cache, the SDM version has higher aggregate IPC than the Seq version because it exploits parallelism, but it slows down even more than Seq once main memory accesses are needed for larger problem sizes. This performance degradation occurs because of the steeper memory wall under SDM due to simultaneous accesses from multiple threads. Thus, *ineffectively parallelized code can actually run slower on a parallel microprocessor*.

To alleviate the problem, we proposed the *Synchronized Pipelined Parallelism Model* (SPPM), an execution paradigm that uses the shared cache as a high-speed communication channel between producer and consumer pairs. SPPM carefully manages the threads' execution so they communicate via the cache, which significantly reduces the burden on the memory interface and better exploits concurrent execution on parallel microprocessors. Our results [28] show SPPM's advantage on the Pentium 4 with Hyperthreading. We have also found SPPM to be very effective on Core Duo processors and even dual CPU PowerMac G5 symmetric multiprocessor (SMP) systems. However, while SPPM performs well on parallel microprocessors with shared caches, it does not perform as desired when cores have private L2 caches. On current private cache CMPs, the overhead of cache line transfers between the caches offsets the advantages achieved by reducing the burden on the memory interface. In this case, the bottleneck is the cache-coherence protocol and the cache controller.

To understand this surprising behavior, we developed a tool called *C2CBench* to evaluate the performance of the storage controllers at different levels of the memory hierarchy under varying workload conditions and to measure the overhead of maintaining cache coherence in parallel microprocessors. This tool helped us explore the behavior of parallel execution paradigms on parallel microprocessors, and has been invaluable in finding bottlenecks in CMP architectures and systems and exploring ways to exploit their features. To overcome the expensive cache coherence operations on CMPs with private caches, we developed a new execution model called *Polymorphic Threads* (PolyThreads), which supports producer-to-consumer communication while minimizing cache-to-cache traffic.

The paper is organized as follows: Section 2 discusses related research. Section 3 defines the SDM and SPPM models and identifies potential bottlenecks that may affect their performance. It introduces PolyThreads as an improvement over SPPM for architectures with private caches. Section 4 describes our benchmarks and enumerates the target architectures. Section 5 presents the results of architecture evaluation and explains how the programming models are impacted, while Section 6 compares performance of SDM, SPPM, and PolyThreads on the target systems. Finally, Section 7 describes our ongoing research into parallelization of applications for parallel microprocessors.

2 Related Research

SPPM was inspired by I-Structures [4] used in data-flow architectures for fine-grained producer-consumer synchronization. Others have proposed using an I-Structure-like Synchronization Array [25, 24] to decouple producer and consumer threads for pointer chasing code, and target the exploitation of extremely fine-grained parallelism whereas we seek to target much coarser-grained parallelism.

In the past, pipelined computation was the basis for systolic arrays [16, 3] and vector processors Cray-1 [26]. More recently, stream processing has been a topic of research in both the industry and the academia. For example, the *StreamIt* programming language and compilation infrastructure [12, 11] was developed to characterize large streaming applications and efficiently map their concurrent kernels to a number of target architectures.

Locality enhancement techniques [21] are software optimizations applied with the objective of improving the behavior of memory intensive applications. By making maximal usage of data fetched into the cache before it gets evicted, these optimizations improve both the spatial and temporal locality of applications. For example, tiling [29, 6, 7], also known as blocking [15, 18], transforms loop nests whose working set sizes far exceed the cache size to improve the temporal locality by working on smaller tiles of data. The transformation causes a tile to be fully used before it is evicted from the cache.

Research into cache partitioning to avoid interference and to improve shared cache performance on SMT processors for scientific codes with perfect loops [23] uses tiling and copying to reduce capacity and conflict misses. However, modifications to the operating system or access to performance counters are needed to detect interference.

Cache coherent shared memory machines have been around for decades, and nearly every paper evaluating such architectures contains some performance estimations or measurements of the timing or bandwidth. Early machines such as DASH [19] and Flash [17] significantly influenced how future scalable cache coherent machines were designed. One effort to measure NUMA system performance is found in [9]. The authors develop a tool to evaluate link throughput between caches in scalable NUMA machines. Somewhat less was done with commodity, bus-based SMP systems. For example, the authors of [14] evaluate the Pentium Pro processors' performance in great detail but much less detail is provided regarding cache coherence behavior and timing. lmbench [22] provides much detail on many performance metrics, including support for multi-threading contention. However, unlike C2CBench, it does not measure cache-to-cache performance.

Another performance evaluation approach is to use the performance counters available on modern CPUs. The Performance Application Programming Interface (PAPI) [5, 10] tools provide a standardized interface to counters in commodity architectures running Linux. Oprofile [1] is a tool that uses performance counters to profile applications, though with less control than PAPI provides. Both tools aided our efforts to understand SPPM behavior.

3 Parallel Program Execution Models

The *Spatial Decomposition Model* (SDM) exploits data parallelism by partitioning data and associated computation across the available processors. It is relatively easy to program for suitable algorithms and is widely used by both programmers and parallelizing compilers. SDM threads work on their respective data partitions independently of each other, causing the already bandwidth-limited shared memory interface on CMP and SMT processors to be further overburdened with a large number of simultaneous accesses. When the application's working set is larger than the cache, SDM induces a large number of capacity misses that further degrade performance. On systems with shared caches, SDM threads may also cause cache interference, evicting each other's data from the shared cache.

The *Synchronized Pipelined Parallelism Model* (SPPM)[28] was developed to reduce the demand on the memory bus by restructuring suitable programs into producer/consumer pairs that communicate through the cache rather than the memory. For architectures that do not provide a shared cache, the *presumed fast* cache coherence mechanism should still provide a higher bandwidth communication channel than the main memory, which turns out not to be the case (see Section 5). Figure 2 shows a logical representation of how SPPM works. The main memory holds the computation's input data blocks, each of which

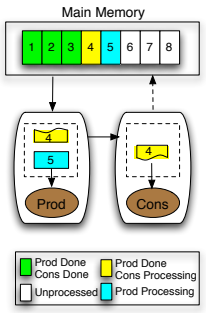

Figure 2. SPPM Execution

is fetched by the producer and possibly modified. The producer's results are then passed on to the consumer for further processing, after which the consumer's results are written back to main memory. The figure shows the different states the data blocks are in at a given instant: some blocks are done, one is being transferred from producer to consumer and being processed by the consumer, while another is being fetched and processed by the producer. While the figure shows a single data array, often several arrays or streams are needed.

Restructuring suitable applications to exploit producer-consumer parallelism is often more complex than using SDM, though some streaming applications are naturally suited to producer-consumer parallelism. Work is underway to ease the programming of SPPM applications (see Section 7). Depending on the architecture, the producer and consumer may communicate using either a shared cache or the cache coherence interconnect connecting the cores/processors. In either case, the number of consumer's memory accesses is greatly reduced, with aggregate memory bandwidth similar to that of the sequential program.

Polymorphic Threads model was developed because SPPM did not perform well on CMPs with private caches. While PolyThreads is derived from SPPM, the communication paradigm is entirely different. Each polymorphic thread contains both the producer and consumer code, which operate on blocks of data, with each thread operating on alternating blocks. When a thread's producer code is finished with a block, it signals the other thread's producer to begin work on the next input block. It also transforms itself into a consumer for the data just produced by itself as a producer, as illustrated in Figure 3. Because the consumer uses data just produced by its producer persona, it finds the data in its local cache, thus no large cache-to-cache transfers are needed. This allows concurrent execution of polymorphic producer-consumer threads, largely communicating results through the local cache. While synchronization still requires the use of the coherence mechanism, such little data is transferred that it does not significantly affect the performance. In addition, the program's access pattern may lead to dependencies that cause a small amount of data to be

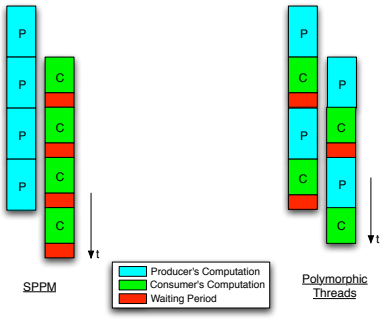

Figure 3. SPPM and Polymorphic Threads

transferred between caches at block boundaries. The block size can be tuned based on the data access pattern and the cache size to achieve the best performance.

Polymorphic Threads greatly benefits SMPs and CMPs that have private caches because cache coherence operations always incur some cost. However, it does not yield any benefit on SMT processors and CMPs with one or more shared cache levels, because producer-consumer communication is already as fast as possible and the overhead of synchronizing and switching modes means it runs slightly slower than SPPM on such processors. In addition, Poly-Threads programs are more complex and difficult to develop because of the fusion of producer and consumer code with the transformation at a synchronization point in the middle. The added complexity and slight additional overhead means that it should only be used when needed to overcome the cache coherence cost of processors having private caches. As described in Section 7, efforts are underway to reduce the burden of programming such code.

4 Benchmarks and Target Architectures

We have currently hand-coded the following four applications as benchmarks for our research. These applications are real-world segments, yet are simple enough to isolate the memory behavior and discover performance differences between the various parallel execution approaches.

Red-Black Equation Solver (Red-Black): This iterative equation solver kernel solves a partial differential equation on a grid using a finite differencing method. Due to the ordering nature of the grid point updates, this algorithm is not parallelizable using SDM. Authors in [8] suggest restructuring the algorithm to update the grid points in a different order, called the *Red-Black Ordering*, that lends itself more easily to parallelization. A detailed description of the program code can be found in [28].

Finite Difference Time Domain Simulation (FDTD): The Finite Difference Time Domain application [27] is an extremely memory-intensive electromagnetic simulation. It computes many time steps over a 3D volume. A detailed description of the program code can be found in [28].

ARC4 Stream Cipher (ARC4): ARC4 is the *Alleged RC4*, a stream cipher commonly used in protocols such as the Secure Sockets Layer (SSL) for data security on the Internet and Wired Equivalent Privacy (WEP) in wireless networks. The encryption process uses a secret user key in a key scheduling algorithm to initialize internal state, which is used by a pseudo-random number generator to generate a keystream of pseudo-random bits. As each byte of the keystream is generated, it is XORed with the corresponding byte of the plaintext to generate a byte of the ciphertext. Because the generation of each byte of the keystream is dependent on the previous internal state, the process is inherently sequential and, thus, non-parallelizable using SDM. By treating the keystream generation and the production of ciphertext as producer and consumer, SPPM and Poly-Threads can exploit the inherent concurrency.

The Pipelined Equation Solver (EQN): The pipelined equation solver (EQN) solves the same sort of problem as Red-Black, described above. Instead of restructuring the original algorithm with its intra-loop dependences, it is pipelined by executing its iterations concurrently. As an iteration produces semantically enough data, a new iteration is spawned on another processor, if available. This forms a linear chain of processors with each acting as a producer for the next one in the chain and as a consumer for the previous one. This form of pipelined computation leads to a few iterations toward the end being executed speculatively. In our implementation, the processor that achieves convergence first signals the other processors to stop. This benchmark, like ARC4 above, is representative of classes of applications that are not parallelizable at all using SDM. Moreover, it demonstrates one way that SPPM can scale with the availability of more than two hardware threads.

Our benchmarks were run on the following mix of CMP and SMT processor based systems.

- 3 GHz Intel Pentium 4 with Hyperthreading and 512K Level 2 cache and 1GB dual-channel PC3200 RAM running Linux 2.6.16 (labeled *P4*)
- 2-way dual-core 2.0GHz Opteron 270 system with private 64K L1 and 1024K L2 caches per core and 16GB dual-channel ECC PC3200 memory running Linux 2.6.16 (labeled *Opteron*).
- Dual-core 2.4GHz Athlon 64 X2 system with private 64K L1 and 512K L2 caches on each core and 2GB dual-channel PC3200 memory running Linux 2.6.16 (labeled *Athlon*).
- 2-way dual-core 2.66GHz Intel Core 2 Xeon 5150 Mac Pro with a private 32K L1 cache per core and shared 4096K L2 caches per chip and 1GB FBDIMM RAM running Mac OS X 10.4.7 (labeled *Xeon*).

- 2GHz Intel Core Duo based iMac with a private 32K Level 1 cache on each core and a shared 2048K Level 2 cache between the cores and 512MB PC2-5300 memory running Mac OS X 10.4.7 (labeled *Core Duo*).

On the P4, Intel's C++ Compiler 7.0 was used, while GCC 4.0 was used on all the other systems. The benchmarks were compiled using the -O3 optimization flag on the P4, Opteron and Athlon and using the -fast optimization flag on the Xeon and Core Duo. The performance tool was compiled using the -O optimization flag on all the systems.

5 Architecture Evaluation Results

Our memory hierarchy communications performance tool (C2CBench) measures the performance of various levels of the memory hierarchy under different workload conditions. It can determine relative throughput and latency of accessing local and remote caches and memory with and without interference from other threads. C2CBench is based on the SPPM runtime system, which provides producer and consumer threads that perform specified operations (reads or writes) on the elements of a data set divided into blocks. Using appropriate values for the data set size and block size, one can test the performance of the L1 cache, L2 cache, and memory controllers for either local or remote accesses. The producer and consumer can be configured to either work in lock-step manner or concurrently.

- When working in lock-step, the producer processes a block of data while the consumer waits for it to finish, thus introducing no interference. In turn, the producer waits for the consumer to finish processing that block before proceeding to the next one. This allows us to measure the baseline performance with no contention for access to the storage controllers.

- When working concurrently, the consumer processes a block of data previously processed by the producer while the producer is processing the next block of data. Working concurrently puts additional burden on storage controllers and allows us to measure the degradation in their performance due to contention.

C2CBench also controls buffer placement on NUMA architectures by using CPU affinity and Linux's data placement policies, so the buffer can be placed in physical memory attached to a specified processor. This allows us to measure the effects of local and remote data placement.

5.1 Memory Interface Performance

Figure 4 shows the time in microseconds taken to read each cache line in a 32M block of data from the memory

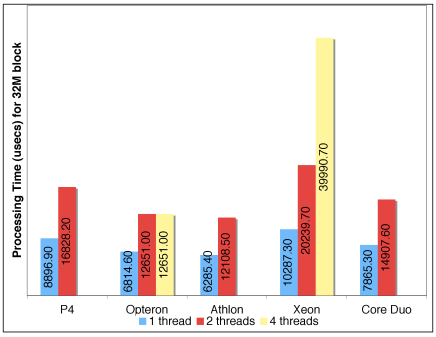

Figure 4. Memory Interface Performance

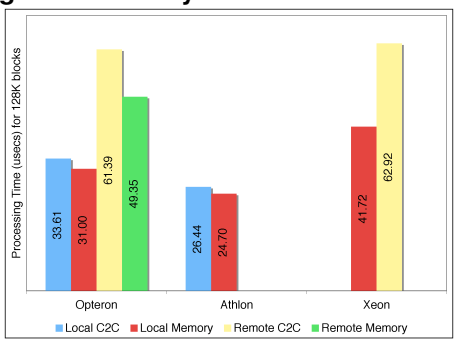

Figure 5. Memory and L2 Performance

with simultaneous accesses by 1, 2 and 4 (where applicable) threads on each architecture. On all the architectures, the performance *decreases* as contention increases. On the Opteron, however, the performance is the same for both 2 and 4 threads because each chip in the system has its own integrated memory controller. This test models the simultaneous accesses to the memory interface in applications parallelized using SDM. The figure clearly shows that memory intensive SDM programs experience degraded performance due to excessive contention for the memory interface.

5.2 Memory and Private L2 Performance

Figure 5 compares the times taken to read a 128K block of data from the following locations: the private cache of another core on the same chip (labeled *Local C2C*); the local memory attached to the chip or frontside bus (labeled *Local Memory*); the cache of a core on a different chip (labeled *Remote C2C*); or the memory attached to another chip (labeled *Remote Memory*). Only the Opteron architecture has all categories, being a NUMA architecture. P4 and Core Duo are not shown because they have only shared L2 caches. The Xeon's memory is classified as Local Memory, though it is shared between all the processors.

The results are surprising, yet confirm our observations of SPPM performance: on the dual-core Opteron and Athlon 64 X2, it is faster to read from local memory than from the other core's cache. On the Opteron, it is also faster

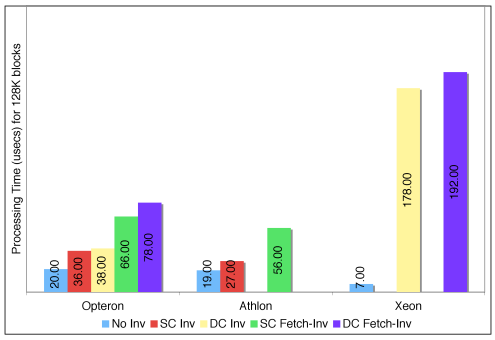

Figure 6. L2 Cache Coherence Overhead

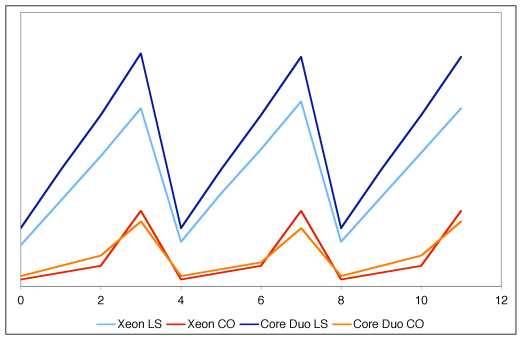

Figure 7. L1 Cache Coherence Overhead

to read from remote memory than from a cache on the other chip. Much of this is due to the fast on-chip memory controllers in these architectures, but the poor cache coherence performance is unexpected. Because SPPM uses the cache to communicate, its performance on these chips is worse than using the memory. Thus SDM actually works better, though it may be slower than sequential code. This test is a useful indicator of SPPM's performance on a target architecture, since many of the consumer's data accesses are satisfied from the producer's cache.

5.3 L2 Cache Coherence Overhead

Figure 6 compares the time taken to write to a 128K block of data resident in the local cache for each of the coherence conditions below. The Core Duo and P4 are not included, because they have shared caches and do not use a coherence protocol (the L1 does on the Core Duo, but a 128K block does not fit into it).

Block is in *Modified* state (labeled *No Inv*): Writing to the data block is cheapest in this case as no invalidations need be sent to the other caches in the system. We use this as the baseline for comparison with the other cases.

Block is in *Shared* state (MESI and MOESI) or *Owned* state (MOESI only): Writing to the data block in these states is more expensive because it involves invalidating other caches in the system that might have a copy of the block. The figure shows two cases: one, where the core whose copy is to be invalidated is on the same chip as the invalidating core (labeled *SC Inv*); the other, where the two cores are on different chips (labeled *DC Inv*). Different chip values are not shown for the Athlon, which only has a single chip, nor are same chip values shown for the Xeon, which shares the L2 cache between same-chip cores.

Block is not resident in cache or has been invalidated: C2CBench does a *read-modify* operation on the block. As a result, it involves fetching the block from the remote cache, after updating its status to *Shared/Owned* there, and then invalidating the remote copy. Thus, this is even more expensive than the previous case, where the remote copy only need to be invalidated, but not fetched. Again, we differentiate between the cases where the source and destination cores are on the same chip (labeled *SC Fetch-Inv*) or different chips (labeled *DC Fetch-Inv*).

As expected, the Fetch-Inv cases are more expensive than invalidation alone, and different-chip operations are more expensive than same-chip ones. In all cases, invalidation time is a large percentage of the fetch and invalidate time, thus invalidation overhead is as expensive as the transfer of cache lines. The Xeon shows high performance when invalidation is not required and very poor performance when it is, with the cache line transfer time nearly negligible compared to the invalidation time. These results show how expensive coherence operations are on these architectures, which directly affects the performance of parallel applications that share data and synchronize often. It is an indication that these parallel microprocessors are not well suited to many types of parallelism in common use.

5.4 L1 Cache Coherence Overhead

This test measures the cache coherence protocol overhead between L1 caches on shared L2 cache architectures (in our case, Core Duo and Xeon). Figure 7 shows the processing times for each 64K block of a 256K data set read by the consumer over three iterations of the data set on the Core Duo and Xeon. The processing times of the individual blocks are cumulatively added across an entire iteration, which gives rise to the *saw-tooth* like waveform. The tests are run with the producer and consumer running concurrently (CO) and in lock-step (LS). The size of Level 1 caches on both architectures is 32K, and they apparently use a write-back policy. When executing in lock-step with the producer, the consumer always finds the first 32K of a 64K block in the shared Level 2 cache, where it must have been evicted from the producer's cache to accommodate the last 32K of the block. The last 32K of the 64K block must, therefore, be read from the producer's Level 1 cache, apparently by first causing it to be written back to the shared Level 2 cache. As a result, reading the last 32K is more expensive than reading the first 32K, and this additional expense has to be incurred for every block read by the con-

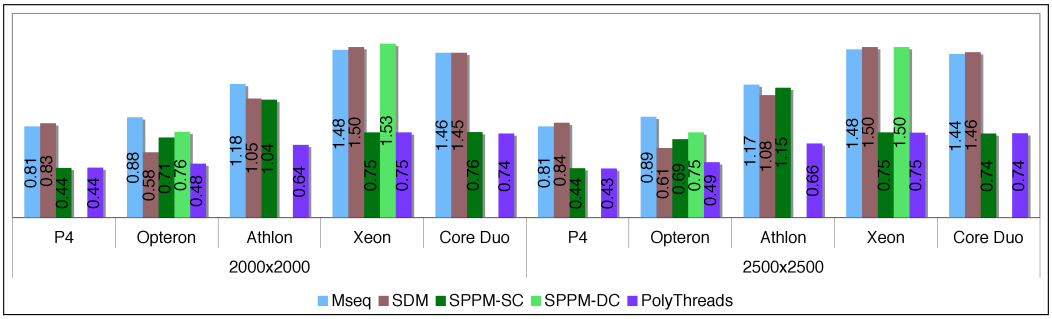

Figure 8. Red-Black Normalized Execution Times

sumer. This explains the linear waveform corresponding to lock-step execution. However, when the producer and consumer execute concurrently, the producer does not wait for the consumer to finish reading every block, and keeps evicting newer blocks to the Level 2 cache even as the consumer is reading older blocks that were evicted before. In this case, all but the last 32K of the last block are evicted from the producer's Level 1 cache, and as can be seen from the figures, the consumer processes them very fast. But reading the last 64K block takes as much time as when executing in lock-step, which causes the spike seen in its waveform. Overall, the consumer performs better when executing concurrently than when executing in lock-step. This test shows that extremely fine-grained data sharing between threads can harm performance, so it is important that the granularity be larger than the L1 cache size. We show this behavior in SPPM for the Red-Black benchmark in Section 6.

6 Programming Model Evaluation Results

We now present the performance measurements of the Red-Black, FDTD, ARC4, and EQN applications using the SDM, SPPM and *Polymorphic Threads*.

6.1 Red-Black Equation Solver

Figure 8 shows normalized execution times of the modified sequential (MSeq) version [8], the SDM, SPPM, and PolyThreads versions for two problem sizes (2000×2000 and 2500×2500). The SPPM-SC version is SPPM with the producer and consumer threads running on the *S*ame *C*hip (SC) while SPPM-DC is SPPM with the producer and consumer threads running on *D*ifferent *C*hips. All values in the figure are normalized to the corresponding sequential version execution time value on each machine. Because of the red-black access pattern, MSeq incurs increased memory references over the sequential version, which does not need the red-black access pattern, and is usually slower. All the parallel versions are based on MSeq's access pattern. The figure shows that SPPM is faster than SDM on all the shared cache machines (P4, Core Duo, and Xeon). The SDM version is as fast or faster than the normal SPPM versions on the AMD architectures due to the high bandwidth integrated memory controller and the high overhead of the cache coherence protocol demonstrated earlier. On these architectures, however, PolyThreads is significantly faster than the other versions, because it greatly reduces the cache-to-cache transfers. In fact, it is even as fast as SPPM on the Intel architectures, so it does not induce any performance penalty in Red-Black.

As we demonstrated in Figure 7, on shared architectures with private L1 caches, there is an overhead involved for maintaining cache coherence when data is dirty in the L1 caches. This overhead penalizes applications with extremely fine-grained data sharing. In Figure 9, we vary this granularity for a problem size of 1000×1000 in the SPPM version of the Red-Black equation solver on the Xeon and the Core Duo. Granularity is measured in number of rows of the input array i.e. 1000 data elements ($\approx$ 8KB of double precision floating point numbers). The size of the L1 cache on both the Xeon and Core Duo is 32KB. For granularities finer than 4 rows ($\approx$ 32KB), the consumer incurs the high overhead of the L1 cache coherence protocol while fetching data from the producer's L1 cache, causing the overall execution time to increase. As the granularity is increased to greater than 4 rows, fewer data elements are fetched from the producer's L1 cache by the consumer, so the execution time decreases, hence performance increases (more pronounced on the Xeon than on the Core Duo). Al-

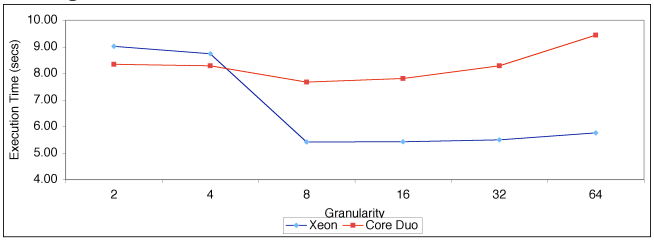

Figure 9. Red-Black – Effect of granularity

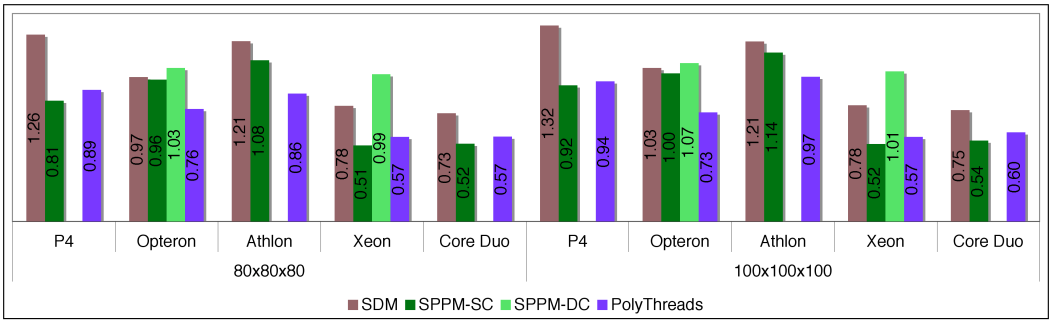

Figure 10. FDTD Normalized Execution Times

though there is a loss in parallelism with increasing granularity, the loss is more than compensated for by not having to fetch a prohibitive amount of data from the producer's L1 cache. However, for very coarse levels of granularity at the right of the figure, the loss in parallelism can no longer be overcome, so performance degrades.

6.2 FDTD

Figure 10 shows normalized execution times of the SDM, SPPM and PolyThreads versions of FDTD on the target architectures for two problem sizes ($80 \times 80 \times 80$ and $100 \times 100 \times 100$). SPPM-SC and SPPM-DC are defined in the same way as in Red-Black above. All values in the figure are normalized to the corresponding sequential program value on each machine. Because SPPM makes better use of the cache, it performs better than SDM in almost all cases, but is particularly effective on shared cache architectures. Because FDTD accesses so many arrays per iteration, it is extremely data-intensive, thus the SDM version tends to run much more slowly than the sequential version on the P4. Despite the sharing of resources by HyperThreading on the P4, SPPM still manages to extract a significant performance improvement. SPPM also achieves nearly perfect speedup on the Core Duo and Xeon. SPPM-SC ekes out a performance advantage over SDM on the AMD chips, but is not near a perfect speedup, while SPPM-DC is as slow as the sequential version or worse. PolyThreads, which reduces the cache-to-cache transfers, does provide better speedup on Athlon and Opteron, though still not as good as on the Intel chips. It is slower than SPPM on the shared cache architectures because of its slight overhead.

6.3 ARC4 Encryption

Figure 11 compares the encryption throughput of the sequential (Seq), SPPM (Same Chip and Different Chips), and PolyThreads versions (higher values are better). ARC4's producer and consumer are asymmetric, so the producer

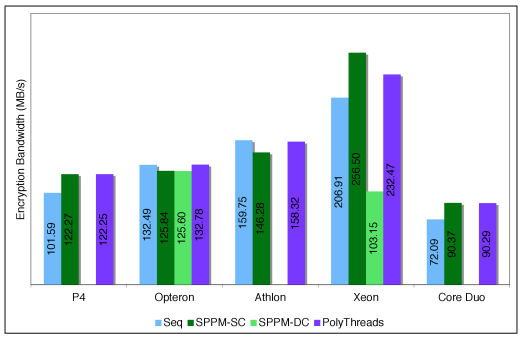

Figure 11. ARC4 Encryption Bandwidths

(key generator) does much more work than the consumer (XOR the streams). SPPM achieves significant speed-up on shared cache architectures, but incurs a performance penalty on the ones with private caches due to the high cache coherence overhead. Even PolyThreads incurs a penalty on the AMD architectures. Because of the high memory bandwidth on these architectures, the sequential version performs roughly as well as PolyThreads. Moreover, the fact that the consumer does much less work than the producer naturally limits the speedup achievable in this specific case as can be seen from the figure. Nevertheless, this algorithm represents a class of applications not parallelizable using SDM but which can be run in parallel using SPPM and PolyThreads.

6.4 Pipelined Equation Solver (EQN)

Figure 12 shows the normalized execution times of EQN on the target architectures for varying number of threads. The execution times are normalized to the execution time using a single thread (1T). 2T and 4T are the execution times using 2 and 4 threads respectively. In all the cases, SPPM scales nearly linearly from 1 to 2 threads. On the Opteron, performance does not scale linearly while going from 2 to 4 threads. When running 4 threads, 2 threads running on different chips have to share data between them. The higher overhead of the cache coherence protocol be-

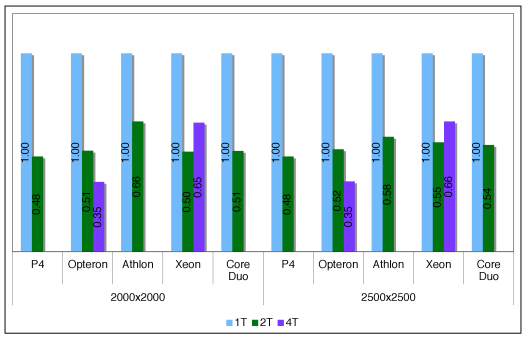

Figure 12. EQN Normalized Execution Times

tween them forms the bottleneck in the system and results in a less than linear speed-up. On the Xeon, going from 2 to 4 threads is causes a degradation in performance. This is a result of the extremely high overhead of maintaining cache coherence across chips (see Figure 6) as opposed to no overhead within a chip due to the shared cache.

As a result of these experiments, it is clear that SPPM does very well when the producer and consumer share a cache. It is also clear that the communication mechanism in SPPM is penalized by slow cache coherence mechanisms, so *Polymorphic Threads* must be used instead for good performance on architectures with private caches. The results also show that the conventional parallel execution model i.e., SDM can often run more slowly than the sequential version on current parallel microprocessors, a problem which motivated this research in the first place. Finally, the results show one way that SPPM can scale to more than two threads/processors.

7 Ongoing Work

While SPPM and *Polymorphic Threads* are better suited than SDM on modern parallel processors for some classes of data-intensive applications, they are more difficult than SDM in terms of programming effort. To simplify the process of parallelizing applications using SPPM and PolyThreads, we are designing software infrastructure to support exploiting parallelism and managing synchronization. This would allow programmers to avoid tedious hand coding and concentrate on the functional aspects of their applications. This software infrastructure development will be complemented by formulating techniques to model the performance of Seq, SDM, SPPM and PolyThreads in terms of the architectural and model-specific parameters obtained using C2CBench. This will allow programmers to quantitatively compare the different models and to choose what is best for a given application. Efforts are also underway to enhance SPPM and PolyThreads for scalability when more cores are available for use. The eventual goal of this research is to augment parallelizing compilers to identify and exploit SPPM and PolyThreads-style parallelism in suitable applications. We are also evaluating hardware modifications to support high performance communication and synchronization between processors.

8 Conclusion

Writing efficient software for modern parallel microprocessors is an architecture-driven exercise. In adopting a parallel programming model and applying advanced optimizations, attention must be paid to the capabilities of the memory interface and cache controllers in the system and the overhead of the protocol used to maintain cache coherence among the processors. Using C2CBench, we have experimentally evaluated several modern parallel microprocessors based on important criteria that affect performance. We have shown how sustained simultaneous accesses to the memory interface can degrade performance in data-intensive applications. We have also shown that in architectures having private caches, even though the caches and the processor interconnect themselves are capable of sustaining a high data bandwidth, the actual bandwidth available is severely limited by the excessive expense of the cache coherence protocol. Even in architectures with shared caches, data transfer speeds between private caches is still very slow, which assumes importance for choosing the granularity of data sharing between the threads.

We compared the conventional *Spatial Decomposition Model* (SDM) with our *Synchronized Pipelined Parallelism Model* (SPPM) and *Polymorphic Threads* for several different classes of data-intensive applications. On all shared cache architectures, SPPM and PolyThreads perform better than SDM because they put less burden on the memory interface and carefully manage the shared cache for inter-thread communications. For some applications, there is a small penalty in using PolyThreads compared to SPPM due to more synchronization in the code. On private cache architectures, SPPM's performance degrades due to the high overhead of cache-to-cache data transfers. SDM still does not perform well on such architectures because it strains the memory interface. *Polymorphic Threads*, however, improves performs significantly because of the way it localizes inter-thread communication within the same processor, thus avoiding expensive inter-processor data transfers.

Using a performance tool like C2CBench makes it possible to explore the complex design space of parallel execution on modern parallel microprocessors. Bottlenecks found by the tool can be addressed or overcome by programming models and execution infrastructures like SPPM and PolyThreads. Additional programming support for these approaches will allow programmers to focus on functional aspects rather than having to deal with issues of parallelization and management of synchronization. This will lead to more

efficient utilization of the computational, memory, and communication throughput available on current and next generation parallel microprocessors.

References

[1] Oprofile: A system profiler for Linux. http://oprofile.sourceforge.net.

[2] AMD. AMD Athlon 64 X2 Dual-Core Processor Product Data Sheet, 2006. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/33425.pdf.

[3] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, and M. Lam. "The Warp Computer: Architecture, Implementation, and Performance". *IEEE Trans. Comput.*, 36(12):1523–1538, 1987.

[4] Arvind and R. E. Thomas. "I-Structures: An Efficient Data Type for Functional Languages". Technical Report LCS/TM-178, MIT laboratory for Computer Science, 1980.

[5] S. Browne, C. Deane, G. Ho, and P. Mucci. PAPI: A Portable Interface to Hardware Performance Counters. In *Proceedings of Department of Defense HPCMP Users Group Conference*, June 1999.

[6] J. Chame and S. Moon. "A tile selection algorithm for data locality and cache interference". In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 492–499. ACM Press, 1999.

[7] S. Coleman and K. S. McKinley. "Tile size selection using cache organization and data layout". In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 279–290. ACM Press, 1995.

[8] D. E. Culler, A. Gupta, and J. P. Singh. *"Parallel Computer Architecture: A Hardware/Software Approach"*. Morgan Kaufmann Publishers Inc., 1997.

[9] A. Davis and U. Prestor. An application centric cc-numa profiler. In *Proceedings of the IEEE Workshop on Workload Characterization IV*, 2001.

[10] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using PAPI for Hardware Performance Monitoring on Linux Systems. In *Conference on Linux Clusters: The HPC Revolution*, Linux Clusters Institute, Urbana, Illinois, 2001.

[11] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS 06: Proceedings of the twelfth international conference on Architectural support for programming languages and operating systems*, October 2006.

[12] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS 02: Proceedings of the tenth international conference on Architectural support for programming languages and operating systems*, 2002.

[13] Intel. Intel Core Duo Processor Product Brief, 2006. http://www.intel.com/products/processor/coreduo/product-brief.pdf.

[14] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance characterization of a quad pentium pro SMP using OLTP workloads. In *ISCA*, pages 15–26, 1998.

[15] I. Kodukula, N. Ahmed, and K. Pingali. "Data-centric multi-level blocking". In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 346–357. ACM Press, 1997.

[16] H. T. Kung and C. E. Leiserson. "Systolic Arrays (for VLSI)". Technical Report CS 79-103, Carnegie Mellon University, 1978.

[17] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. "The Stanford Flash Multiprocessor". In *Proceedings of 21st International Symposium on Computer Architecture*, Chicago, IL, 1994.

[18] M. D. Lam, E. E. Rothberg, and M. E. Wolf. "The cache performance and optimizations of blocked algorithms". In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 63–74. ACM Press, 1991.

[19] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. "The Dash Prototype: Implementation and Performance". In *Proceedings of ACM SIGPLAN '91 COnference on Programming Language Design and Implementation*, Gold Coast, Australia, 1992.

[20] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. "Hyper-Threading Technology Architecture and Microarchitecture". *Intel Technology Journal*, 6(1):4–15, February 2002.

[21] K. S. McKinley, S. Carr, and C.-W. Tseng. "Improving data locality with loop transformations". *ACM Transactions on Programming Language and Systems*, 18(4):424–453, 1996.

[22] L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.

[23] D. S. Nikolopoulos. "Code and Data Transformations for Improving Shared Cache Performance on SMT Processors". In *Proceedings of the 5th International Symposium on High Performance Computing*, Tokio-Odaiba, Japan, 2003.

[24] G. Ottoni, R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. "Decoupled Software Pipelining: A Promising Technique to Exploit Thread Level Parallelism". In *Proceedings of the Fourth Workshop on Explicitly Parallel Instruction Computer Architectures and Compiler Technology*, March 2005.

[25] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. "Decoupled Software Pipelining with the Synchronization Array". In *PACT '04: 13th International Conference on Parallel Architectures and Compilation Techniques*, page 177. IEEE Computer Society, 2004.

[26] R. M. Russell. "The Cray-1 Computer System". *Communications of the ACM*, 21(1):63–72, January 1978.

[27] A. Taflove. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House, Boston, 1995.

[28] S. N. Vadlamani and S. F. Jenks. "The Synchronized Pipelined Parallelism Model". In *The* 16^{th} *IASTED International Conference on Parallel and Distributed Computing and Systems*, Cambridge, MA, USA, 2004.

[29] M. Wolfe. More Iteration Space Tiling. In *Proceedings of Conference on Supercomputing*, pages 655–664, Reno, NV, 1989.