Speculative Precomputation on Chip Multiprocessors

Jeffery A. Brown[1,2], Hong Wang[1], George Chrysos[3], Perry H. Wang[1], and John P. Shen[1]

Microprocessor Research[1] Intel Labs — Computer Science[2] UC San Diego — Massachusetts Microprocessor Design[3] Intel Corp

Abstract

Previous work on speculative precomputation (SP) on simultaneous multithreaded (SMT) architectures has shown significant benefits. The SP techniques improve single-threaded program performance by utilizing otherwise idle thread contexts to run "helper threads", which prefetch critical data into shared caches and reduce the time the "main thread" stalls waiting for long latency outstanding loads. This technique effectively exploits the parallel thread contexts and the data cache sharing at all levels of the memory hierarchy that SMT provides. Chip multiprocessor (CMP) architectures also feature parallel thread contexts, but do not share caches near execution resources. In this paper, we first investigate SP on a basic CMP and show that while the existing SP techniques can provide performance improvements for single-threaded application on such CMP architectures, they fall short of the benefits provided on SMT architectures due to the reduced degree of cache sharing. We then propose and evaluate several simple enhancements to the basic CMP architecture, which can increase the speedup from using SP by an additional 10 to 12%.

1. Introduction

Memory latency has become the critical bottleneck in achieving high performance on modern processors. Many large applications today are memory intensive, because their memory access patterns are difficult to predict and their working sets are becoming quite large. This problem worsens when executing *pointer-intensive* applications, which tend to defy conventional stride-based prefetching techniques.

Speculative Precomputation (SP) is a thread-based cache prefetching technique, which has been introduced in various forms [33, 22, 8, 7, 18, 19, 31, 32] to speed up memory intensive single-threaded applications. The key idea behind SP is to utilize otherwise idle hardware thread contexts to execute speculative threads, as *helper threads*, on behalf of the *main* (non-speculative) thread. These speculative threads attempt to trigger future cache-miss events far enough in advance of cache accesses by the non-speculative thread that the memory miss latency can be masked. SP can be thought of as a special prefetch mechanism that effectively targets load instructions that exhibit unpredictable irregular or data-dependent access patterns. Traditionally, these loads have been difficult to handle via either hardware prefetchers [5, 14, 13] or software prefetchers [20]. For most programs, only a small number of poorly behaving static loads, called *delinquent loads*, are responsible for the vast majority of cache misses [1]. To perform effective prefetch for delinquent loads, SP requires the construction of the *precomputation slices*, or p-slices, which consist of dependent instructions that compute the addresses accessed by delinquent loads. When an event triggers the invocation of a p-slice, a speculative thread, spawned to execute the p-slice, prefetches for the delinquent load that will be executed later by the main thread.

SP was initially introduced on simultaneous multithreaded (SMT) architectures [30, 9, 11], which allow sharing of both execution and memory resources between multiple hardware threads within a single cycle. Because of the communication mechanism on which SP relies, that is, shared caches in all levels of the memory hierarchy, SMT is an attractive multithreading model for supporting SP. Given a sufficient head start, SP helper threads can prefetch critical-path delinquent data closer to the processor core, and even make the data available at the first-level data cache by the time the main thread needs it.

In this research we consider the SP technique on chip multiprocessor (CMP) architecture models [21, 10, 6, 2, 15]. Similar to SMT, CMP provides multiple hardware contexts and the ability to execute instructions from multiple threads within a single cycle. However, unlike SMT where both execution resources and entire cache hierarchies are shared among threads, the CMP processors achieve thread-level parallelism with independent and replicated collections of execution core resources. For memory resources, each core has its own local caches (e.g. L1 and L2) and multiple cores only share the lower[†] level of the memory hierarchy (e.g. L3). While CMP models offer less contention and

[†]For descriptive clarity, we adopt the convention that L1 caches are the highest-level caches, and others are at successively lower levels.

lower implementation complexity, they do so at the cost of inter-thread communication latency. Since the data sharing on CMP is at lower levels of cache hierarchy, the sharing incurs much longer latency than that between two threads on SMT. This is because the farther away from the cores and closer to memory, the longer the latency. In this paper, the baseline CMP model, called *basic CMP*, consists of multiple identical processor cores, each having its private L1 and L2 caches. All cores together share a common L3 cache. In contrast, a SMT can accommodate such sharing in the L1 and L2 caches as well.

On a CMP, SP can be applied by using one core to run the main thread and the other cores for helper threads to do precomputation and prefetches for the main core, thus turning these otherwise idle cores into helper cores. In this paper, we first demonstrate that existing SP techniques, even when applied to the basic CMP directly, can provide non-trivial performance improvements for the memory-intensive single-threaded workloads. In light of SP performance on SMT, we further quantitatively confirm the intuition that the inter-core resource independence (i.e. lack of sharing of resources closer to the core) in CMP can become a hindrance to realize the full benefit of the SP techniques. In particular, since the SP helper threads usually perform simple computation leading to address resolution followed by a prefetch, its execution does not demand much execution resources. In addition, the effectiveness of SP depends heavily on timely sharing of caches in order to assist the main thread. On a basic CMP where cores only share L3, the timeliness can be hindered due to a much higher latency to access prefetched data at L3. To further improve the performance of SP on CMP processors, we then propose a set of simple optimization techniques to further improve the performance of SP on CMP processors by approximating the effect of high-level cache sharing that SMT provides.

It is important to note that this research does not purport to compare the relative merits of SMT over CMP, or vice-versa. Such a comparison cannot be made fairly without a careful examination of how resources should be distributed in order to declare an SMT processor and a CMP processor "comparable". Instead, since the tradeoffs leading to the performance improvement by SP are well understood, our interest in SP on SMT is primarily to draw comparative insights to help reason the tradeoffs that can lead to performance improvement for CMP. In particular, the questions of interest to our work are: "Given a CMP processor with a fixed number of cores and a single thread to execute, how well can SP use otherwise idle cores to improve performance, relative to single-core execution on this same processor? How many cores should we utilize?"

The rest of the paper is organized as follows. In Section 2, we review other speculative multithreading research works that have explored CMP resources for prefetches. In Section 3, we describe details of the CMP machine model and performance evaluation methodology. Section 4 presents the performance evaluation for SP on the basic CMP. In Section 5, a few simple optimization techniques are introduced to enhance SP performance on CMP. Section 6 concludes.

2. Related Work

Prior work regarding using CMP to speedup single-threaded application performance can be categorized into two main areas, *thread level speculation* and *slipstream speculation*.

First, thread level speculation (TLS) on CMP [24, 26, 16, 28] is an aggressive technique that attempts to take a single-threaded program and arbitrarily break it into a sequenced group of threads that may be run in parallel. To ensure that the parallelized program executes correctly, additional hardware must be used to track all inter-thread data dependencies. When a younger thread in the sequence causes a true dependence violation by reading data too early, the hardware must ensure that the mis-speculated thread or the subset of instructions depending on the mis-speculated data are re-executed with the right data. SP differs from TLS in that the execution of SP helper threads does not affect the correctness of the main thread and therefore SP does not require any additional complicated hardware for inter-thread dependency checking and mis-speculation recovery.

Second, the slipstream speculation paradigm [27] is proposed to explore the assumption that only a subset of the original dynamic instruction stream is needed to make full, correct and forward progress. In slipstreaming, the OS redundantly instantiates two copies of a user program, each having its own context. The two duplicated programs run simultaneously on a single CMP. One of the programs (so-called A-stream) is always running slightly ahead of the other (so-called R-stream). Special hardware is used to monitor the trailing R-stream and detects dynamic instructions that predictably have no observable effects and dynamic branches whose outcomes are consistently predicable. Future instances of ineffectual instructions, predictable branch instructions and dependency chain leading up to them, are speculatively bypassed in the A-stream, which is thus sped up. Compared to TLS, slipstream will run R-stream non-speculatively thus effectively ensuring correctness won't be affected by the speculation done in the A-stream. However, to accurately and timely track inter-thread progress, identify ineffectual instructions, coordinate interesting branches across CMP cores, and synchronize relevant events in timely manner, the monitor hardware will not only incur significant design complexity but also become a performance bottleneck affecting the accuracy or timeliness of the monitored events. In contrast to slipstreaming, SP does

Pipeline Structure	12 stage pipeline, 2 cycle misfetch penalty, 10 cycle mispredict penalty
Fetch	2 bundles per cycle
Branch Predictor	2K entry GSHARE 256 entry 4-way associative BTB
Expansion Queue	Private, per-thread, in-order 8 bundle queue
Register Files	Private, per-thread register files. 128 Integer Registers, 128 FP Registers, 64 Predicate Registers, 128 Control Registers
Execute Bandwidth	Up to 6 instructions from one thread
Cache Structure	L1 (separate I and D): 16K 4-way, 8 way banked, 1 cycle latency L2 (unified): 256K 4-way, 8 way banked, 14 cycle latency L3 (unified): 3072K 12-way, 1 way banked, 30 cycle latency All caches have 64 byte lines. Data caches are write-back and write-allocate.
Memory Latency	230 cycle latency, TLB Miss Penalty 30 cycles

Table 1. Details of a research in-order Itanium processor core in the CMP model

not require any additional hardware support beyond what has been implemented in the stock CMP designs [6, 2, 15].

To our knowledge, prior to this paper, there has been no known published work that applies SP to CMP and sheds light through quantitative analysis on the key tradeoffs, especially those that are unique to CMP.

3. Experimental Methodology

3.1. CMP Core Processor Microarchitecture

This paper studies the effects of Speculative Precomputation on a research CMP processor implementing the Itanium Processor Family (IPF) [12] instruction set architecture. In the processor core, instructions are issued in-order, from an eight-bundle expansion queue [23], which operates like an in-order instruction queue. The maximum execution bandwidth is six instructions per cycle, which can be fetched from up to two bundles. Sufficient functional units exist to guarantee that any two issued bundles are executed in parallel without functional unit contention and up to four loads or stores can be performed per cycle.

Each processor core's local memory hierarchy consists of separate 16 KB 4-way set associative L1 instruction and data caches, and a 256 KB 4-way set associative L2 unified cache. We also assume a 3072 KB 12-way associative unified L3. Data caches are multi-way banked, and the instruction cache is single ported. Caches are non-blocking with up to 16 misses in flight at once. A miss upon reaching this limit stalls the execute stage. Speculative threads are permitted to issue loads that will stall the execute stage. A pipelined hardware TLB miss handler [23] is modeled. It resolves TLB misses by fetching the TLB entry from an on-chip buffer that is separated from data and instruction caches. In the default configuration, TLB misses are handled in 30 clock cycles, and we allow memory accesses from speculative helper threads to affect TLB update. In addition, throughout the rest of this paper, we will focus our discussion only on in-order CMP processor models. Albeit beyond the scope of this paper, CMP designs with out-of-order cores can benefit from most, if not all, of the insights of this study. Full details of the modeled processor are shown in Table 1.

3.2. SMT and CMP System Configuration

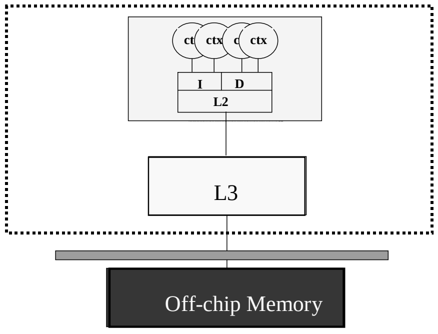
All simulations in this work assume a single non-speculative thread persistently occupies one CMP core throughout its execution while the remaining CMP cores are either idle or occupied by speculative helper threads. Unless explicit distinction is made, the term *non-speculative thread* is used interchangeably with the *main thread* throughout this paper.

For reference, a SMT processor model is depicted in Figure 1(a). It is the baseline processor with the addition of four-way SMT capability. The first thread context is used exclusively to run the main thread; the remaining contexts are used opportunistically for helper thread execution.

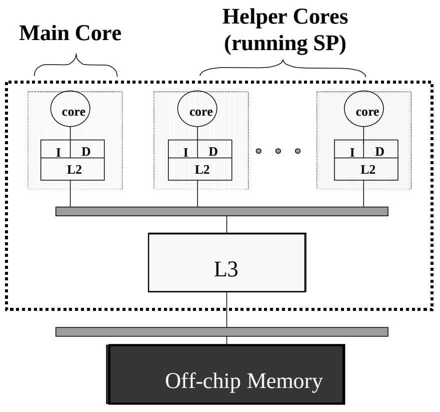
Our CMP processor model is shown in Figure 1(b). It consists of eight processor cores, each a replica of the baseline processor except for the L3 cache. While each core in this CMP has the identical pipeline, identical first- and second-level caches to those of the baseline processor, all cores share a single L3 cache with the same size as that of the baseline through a bus-based interconnect. The first core is used to execute the main thread, and one or more additional cores are made available for helper thread execution. The inter-core interconnect is assumed to be a bus with uniform memory access (UMA) time for each core; since L3 is single-ported, only one core may communicate with the L3 cache at a time, thus bus bandwidth constraints can be accurately modeled simply via contention of the single port on the L3 cache.

3.3. Simulation Environment and Workloads

We model processor performance using SMT-SIM/IPFSim, a version of the SMTSIM simulator [29] that has been enhanced to work with Itanium binaries.



(a) SMT



(b) CMP

Figure 1. SMT vs CMP

Suite	Benchmark	Input	Fast-forward Distance
SPECFP	equake	Training input	1 billion
SPECINT	gzip	Training input	1 billion
SPECINT	mcf	Training input	1 billion
SPECINT	vpr	Training input	1 billion
Olden	health	Max level 5 500 iter	100 million
Olden	mst	1031 graph size	230 million
Olden	treeadd	Breadth-fi rst depth-fi rst	100 million

Table 2. Workload Setup

Benchmark	Number of Slices	Average Length	Average Live-Ins
equake	8	12.5	4.5
gzip	9	9.5	6.0
mcf	6	5.8	2.5
health	8	9.1	5.3
pp-health	2	9.0	3.5
pp-treeadd.df	3	11.3	3.0
pp-treeadd.bf	2	12.5	4.5
pp-mcf	5	14.0	4.4
pp-vpr	6	13.5	4.0

Table 3. Slice Characteristics

SMTSIM/IPFSim is a cycle-accurate, execution-driven simulator of SMT and CMP processors and capable of accurate detailed simulation of resource contention in the cache subsystem, functional units, bus interconnects and branch predictors, just to name a few.

Benchmarks for this study include both integer and floating point benchmarks selected from the CPU2000 suite [25] and pointer-intensive benchmarks from the Olden suite [4]. These benchmarks are selected because either their performance is limited by poor cache performance or they experience high data cache miss rates. The benchmarks and simulation setup are summarized in Table 2. All benchmarks are simulated for 10 million retired instructions after fast-forwarding past initialization code (with cache warmup). In our initial simulation experiments, much longer runs of the benchmarks were performed, however it was observed that the longer-running simulation results yielded only negligible performance differences. In similar treatment as in [8], *gzip* as a compute intensive benchmark is selected as a counter-example.

All binaries used in this work are compiled with the Intel Electron compiler [3, 17] for the IPF architecture. This advanced compiler incorporates the state-of-the-art optimization techniques well known in the compiler community as well as novel techniques specifically designed for the features of the IPF architecture. All benchmarks are compiled with maximum compiler optimizations enabled, including those based on profile-driven feedback, such as aggressive software prefetching, software pipelining, control speculation and data speculation.

For SP helper threads, we employ a mixture of both hand-generated slices as used in [8, 32] and slices automatically generated by an Intel Electron-based post-pass compiler [18] (those with the “pp-” prefix). Table 3 shows some key characteristics of the SP slices used. For each benchmark, there are only a few slices. And on average, these slices are very short (thus creating little contention for execution resources) and have very small numbers of live-ins (implying very small overheads for live-in copies). To help gain insights on potential difference in tradeoffs between SP on SMT and SP on CMP, the same benchmark executables and SP slices previously used in SMT based studies [8, 18, 32] are deployed across all experimental configurations for CMP.

Throughout all of our experiments, the performance metric of interest is the IPC of the main thread. Since the very

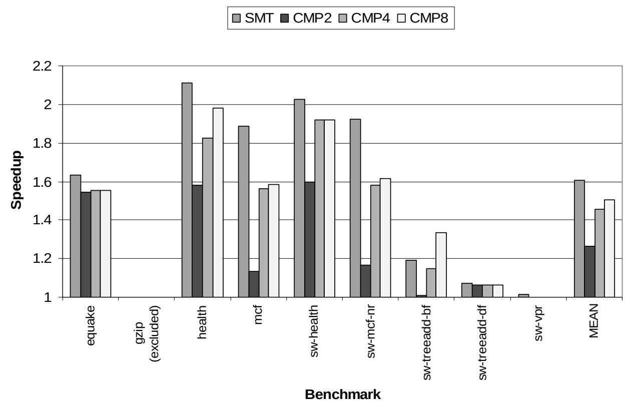


Figure 2. Speedup by SP on Basic CMP

existence of the helper threads is an artifact of our efforts to speed up the main thread, we ignore the IPC of the helper threads. To quantify the relative performance impact of a given experiment, we calculate the speedup for the given experiment as the ratio of the IPC of the experiment to the IPC of the same benchmark running on the single-thread baseline machine, with otherwise identical hardware parameters as described in Section 3.1 and Section 3.2.

4. Performance on Basic CMP

Figure 2 shows the main-thread speedup achieved by SP on the basic CMP over the baseline uniprocessor for the benchmarks under study. For each benchmark, speedups are shown for SP execution on the four-way SMT model, as well as on the *n*-way CMP model with *n-1* cores serving as helper cores running SP helper threads, where *n* = 2, 4, and 8. The far right group shows the arithmetic mean of the speedups for each configuration, over all benchmarks except *gzip*. As shown in Figure 2, *gzip* simply does not benefit from SP on CMP. This is because *gzip*, as a compute intensive benchmark, rarely misses beyond L2 as shown in [8, 32]. SP cannot help such benchmarks whether they are on SMT or CMP. Since SP is detrimental to their performance, one would simply elect not to use SP. Therefore, we've excluded *gzip* from the mean, and from other variations to be considered in the rest of this paper. All reported mean speedups are arithmetic means for all benchmarks except *gzip*.

Figure 2 indicates that across the board, SMT consistently provides the greatest speedup of the four configurations shown, even though it has the fewest overall execution resources and the least amount of aggregate cache capacity. On SMT, the helper threads assist the main thread through prefetching data to caches shared with the main thread at all levels of the cache hierarchy, while doing so at the cost of contending execution resources and memory bandwidth with the main thread. However, the precomputation slices used by helper threads only consume very little execution resources, as indicated in Table 3. The multifold increase of core execution resources in the basic CMP does not necessarily result in helper performance improvement, while the concomitant reduction in cache sharing significantly impacts the effectiveness of the helper prefetches. Due to data sharing at L3 in the basic CMP, even when the main thread hits a prefetch line in the L3, the core still has to endure a relatively long load-to-use latency. By design, the delinquent loads prefetched by the helper threads are likely to be on the critical path of the main thread. Even with high accuracy, the prefetches may not be timely, since those time-critical requests satisfied at L3 are still roughly 30 times slower than those satisfied at L1. As demonstrated in [8], if the top 10 delinquent loads are assumed to always hit at L1, speedups of several factors higher than the uniprocessor performance can be obtained. That limit study indeed quantifies the criticality of the delinquent loads, for which the helper threads are constructed.

Another observation from Figure 2 is that as CMP resources scale up from 2 cores to 8 cores, the performance improvement does indeed scale accordingly. This is due to the high quality of helper thread's slices, which primarily employ *chaining triggers* [8]. Using chaining triggers allows the helper threads to spawn additional helper threads relatively independently from the progress of the main thread. The cost associated with using chaining triggers to spawn helper threads is not taxed on the main thread. Chaining triggering is thus essential for ensuring SP performance scalability on CMP.

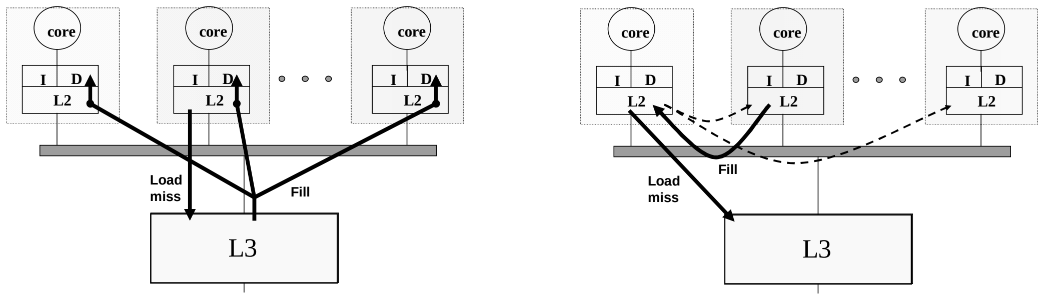
5. Improvements to Basic CMP

As shown in Section 4, while the move to CMP doesn't significantly affect the cost of performing SP, the reduced cache sharing is a detriment and serves as a root cause for the performance gap between SP on CMP and SP on SMT. In order to improve the situation, in this section, we propose and evaluate two schemes to improve the performance of SP on the basic CMP processors. These techniques are geared toward decreasing the latency for the main thread to access data blocks prefetched by the helper cores. Obviously the upper bound of this latency is that incurred by the main thread upon accessing L3 in the basic CMP configuration.

5.1. Improvement 1: Cache Return Broadcast

The first scheme to improve SP performance is called *cache return broadcasting*, or simply *broadcast*. It is depicted in Figure 3(a).

When the shared L3 cache services one core's L2 load misses, the resulting fill is broadcast to all active cores. Given the bus-based UMA interconnect, the physical broad-



(a) Cache Return Broadcast (b) Peer-core L2 Cross-feeding

Figure 3. Two Improvement to Basic CMP for SP

cast already occurs. The proposed hardware change is for the non-requesting cores to accept and process the unsolicited fill, injecting it into their L2 and then L1 data caches. This artificially imposes sharing between the private caches on each core, in an attempt to roughly approximate the sharing that is provided in the SMT model. If successful, this broadcast scheme can effectively reduce the miss rates of the private caches, thereby improving the timeliness of prefetches.

It is important to note that the benefit of broadcast is mutual to both main thread and the helper threads. In particular, the helper cores benefit from this artificial sharing as follows: during periods when the helper threads are not running, the broadcast effectively serves to keep their private caches warm with data used by the main thread. This is important since the helpers often perform loads as part of the slice computation leading to prefetch addresses (e.g. pointer chasing). Without this warmup, the helper threads, after activation, would likely incur cold misses upon using shared variables that have been recently referenced by the main threads. This delay can potentially limit the timeliness of prefetch from the helper threads.

However, it is crucial to note that cache return broadcasting may not be applicable to general workloads; the artificial sharing across cores effectively reduces the aggregate L1 and L2 cache capacity to that of a single core. This effective size reduction can increase both capacity and conflict misses for workloads that don't exhibit a high degree of constructive interference. In SP, helper threads are custom-made to enhance the constructive interference with the main thread's memory references, thus benefiting from this broadcast scheme.

5.2. Improvement 2: Peer-core L2 Cross-feeding

Figure 3 (b) illustrates the second improvement scheme, *peer-core L2 cache cross-feeding*: when one core encounters an L2 miss for load data and submits the request on the bus, like the shared L3 cache, the other active peer cores can snoop on this request and probe their individual private L2 caches to check if they have that data ready to share. If so, the peer core replies to the request and sends the data block to the original requester core. When a transaction occurs, only the lookup time to locate the desired data is affected, presumably resulting in faster service since the L2 lookup time may be shorter than that for L3 lookup; it still takes just as long to transfer the data across the bus and perform the fill in the destination core's private cache hierarchy.

In essence, unlike cache return broadcasting, which loosely simulates a single shared L1 and L2 caches through forced mirroring among disparate cores so as to *reduce the effective miss rates* of the private caches (applicable to both L1 and L2), the peer-core L2 cross-feeding scheme in effect pools the neighbors' L2 caches into a non-inclusive cache, which is looked up in parallel with L3, but returns hits more quickly. This allows a core to utilize some capacity from the neighboring cores, thus *reducing the effective L2 miss penalty* for the shared or prefetched data. In other words, helper cores effectively hoard the prefetched data on their own private caches and deliver them to the main core only when it is needed. While there is potential lack of timeliness as compared to broadcasting scheme, the cross-feeding scheme is unlikely to hurt performance regardless of whether the workloads are latency-bound single-threaded code or throughput-oriented multiprogramming or multitasking code. In practical terms, the cross-feeding mode can be safely left on without the consideration of the actual workload.

Even though the building blocks for realizing such L2 cross-feeding scheme are already present in a typical bus-snooping cache coherence protocol, given a bus interconnect and a snooping-based coherence protocol, the total access time for critical data prefetched by a helper still includes the time for L1 and L2 misses; while service isn't as slow as from L3, it's still far slower than what the L1 cache provides. As with return broadcasting, the scheme is mutually beneficial to both helper and the main threads. The helpers can benefit from this scheme, because it gives them a faster path for performing loads to shared data which is needed to compute prefetches.

5.3. Hybrid of Broadcasting and Cross-feeding

In this research, we also evaluate the effects of using both improvement techniques together. It may seem counter-intuitive at first to combine return broadcasting and cross-feeding: while the latter conceptually pools off-core L2 caches to hoard prefetched data and makes them available to service L2 misses on demand, the former conceptually forces the L1 and L2 caches to be mirrored, which would then make cross-feeding useless. However, the respective appearance of conceptual mirroring and pooling only represents rough approximations, and there is no guarantee that broadcasting will *always* render the cross-feeding useless if used together.

To the contrary, it's plausible that between broadcasts, one core (say P_1) would evict an L2 block holding data (say A) that it later would need while a neighboring core (say P_2) would not and instead retain the private L2 block for its copy of A. In this case, upon P_1's L2 miss due to access to data A, the cross-feeding can supply its copy of A. Since the L2 miss is serviced by P_2 instead of a broadcast data return from L3, each successful cross-feeding transaction represents better utilization of the bandwidth.

5.4. Quantitative Evaluation of CMP Optimizations

To quantify the benefits of these schemes, Figure 4 shows the performance gains from cache return broadcasting and L2 cross-feeding in terms of the mean speedup on our CMP processor, independently and in concert, with two, four, and eight cores active. Note that each of the optimizations achieves additional speedup beyond that of the basic CMP (the leftmost bar in each CMP group). More interestingly, the benefit increases as more cores are activated. In each case, adding cross-feeding to broadcasting increases performance, albeit only by a marginal amount. Broadcasting itself indeed seems to subsume cross-feeding, yet the seemingly less useful cross-feeding doesn't degrade performance. While the eight-core CMP with these improvements achieves a marginally higher speedup than the one-core SMT, it's important to recall that this ensemble has eight times the execution resources and aggregate L1 and L2 cache capacity, so the two configurations aren't directly comparable once the amount of resources is taken into account. With these simple enhancements to the basic CMP architecture, the average speedup from using SP can be improved by an additional 10 to 12%.

One rather unexpected result is that cross-feeding alone provides the most speedup with eight cores active, even exceeding that achieved by the broadcasting scheme. Intuitively, this would seem to be the result of the interconnect being unable to accommodate broadcast traffic with eight

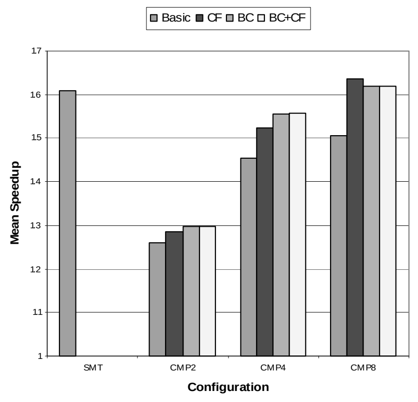


Figure 4. Effects of CMP Optimizations

cores active, while point-to-point cross-feeds utilize it more efficiently. However, this is not an issue. Since the interconnect is a single-channel bus, the broadcasting itself does not cause any additional traffic. In addition, the helper threads are short and not data-memory intensive compared to regular workloads. Cross-feeding outperforms broadcasting with eight cores active because the forced mirroring effect of broadcasting effectively hampers helpers. With some helper threads running far ahead of the remaining threads, data return from trailing threads contend for cache resources at all levels, even on cores running the leading threads with fills for data that are no longer relevant to their execution. For cores running these leading threads, the logically obsolete data are either contained in local cache already, or will evict current useful data that are needed for these threads to make further progress. In short, the indiscriminate UMA broadcasting can effectively let the laggards slow down the pioneers. Furthermore, for the cross-feeding scheme, any additional core that is activated adds to the effective capacity of virtual cache aggregate, which on average has faster service time for L2 misses than that from L3.

6. Conclusion

In this paper, we demonstrate that existing SP techniques provide performance improvements even on basic CMP architectures, but they fall short of their full potential due to the lack of sharing at the levels of cache closer to the execution resources, especially in comparison to performance of SP on SMT. While individual threads on CMP cores incur less resource contention, our evaluation shows that the cost of inter-core communication far outweighs the benefit of decreased execution resource contention. This motivates novel techniques to mitigate this communication bottleneck to further improve SP performance.

We also demonstrate two simple techniques, L3 cache return broadcasting and L2 cache cross-feeding, which can indeed hide some inter-core communication latency and achieve improvement in SP performance. In particular, we find the benefits from SP scale up as more cores are devoted to the SP threads. This not only shows the importance of improving inter-thread communication but also the innate advantage of chaining triggering, which allows helper threads to tolerate inter-thread communication latency, thus fundamentally ensuring scalability of the SP paradigm.

References

[1] S. G. Abraham and B. R. Rau. Predicting Load Latencies using Cache Profiling. Technical Report HPL-94-110, HP Labs, Dec 1994.

[2] 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 *27th International Symposium on Computer Architecture*, 2000.

[3] J. Bharadwaj and et al. The Intel IA-64 Compiler Code Generator. *IEEE Micro*, Sept-Oct 2000.

[4] M. C. Carlisle. Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines. Technical Report PhD Thesis, Princeton University Department of Computer Science, June 1996.

[5] T. Chen. An Effective Programmable Prefetch Engine for On-chip Caches. In *28th International Symposium on Microarchitecture*, Dec 1995.

[6] L. Codrescu and D. S. Wills. Architecture of the Atlas Chip-Multiprocessor. In *International Conference on Computer Design*, 1999.

[7] J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic Speculative Precomputation. In *34th International Symposium on Microarchitecture*, December 2001.

[8] J. Collins, H. Wang, D. Tullsen, H. C, Y.-F. Lee, D. Lavery, and J. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *28th International Symposium on Computer Architecture*, July 2001.

[9] J. Emer. Simultaneous Multithreading: Multiplying Alpha's Performance. In *Microprocessor Forum*, Oct 1999.

[10] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, Mar-Apr 2000.

[11] G. Hinton and J. Shen. Intel's Multi-Threading Technology. In *Microprocessor Forum*, Oct 2001.

[12] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the IA-64 Architecture. *IEEE Micro*, Sept-Oct 2000.

[13] D. Joseph and D. Grunwald. Prefetching using Markov Predictors. In *24th International Symposium on Computer Architecture*, June 1997.

[14] N. Jouppi. Improving Direct-mapped Cache Performance by the Addition of a Small Fully associative Cache and Prefetch Buffers. In *17th International Symposium on Computer Architecture*, May 1990.

[15] J. Kahle. The IBM Power4 Processor. In *Microrpocessor Report*, Oct 1999.

[16] S. Keckler and et al. Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor. In *25th International Symposium on Computer Architecture*, 1998.

[17] R. Krishnaiyer and et al. An Advanced Optimizer for the IA-64 Architecture. *IEEE Micro*, Nov-Dec 2000.

[18] S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. Shen. Post-Pass Binary Adaptation for Software-Based Speculative Precomputation. In *ACM Conference on Programming Language Design and Implementation*, June 2002.

[19] C. K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *28th International Symposium on Computer Architecture*, June 2001.

[20] T. Mowry and A. Gupta. Tolerating Latency through Software-controlled Prefetching in Shared-memory Multiprocessors. *Journal of Parallel and Distributed Computing*, June 1991.

[21] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1996.

[22] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. In *7th IEEE International Symposium on High Performance Computer Architecture*, Jan 2001.

[23] H. Sharangpani and K. Aurora. Itanium Processor Microarchitecture. *IEEE Micro*, Sept-Oct 2000.

[24] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *22nd International Symposium on Computer Architecture*, 1995.

[25] SPEC CPU2000 Documentation. http://www.spec.org/osg/cpu2000/docs/.

[26] J. G. Steffan and T. Mowry. The Potential for Using Thread-level Data Speculation to Facilitate Automatic Parallelization. In *4th IEEE International Symposium on High Performance Computer Architecture*, 1998.

[27] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov 2000.

[28] M. Tremblay. MAJC: An Architecture for the New Millennium. In *Hot Chips 11*, 1999.

[29] D. M. Tullsen. Simulation and Modeling of a simultaneous multithreaded processor. In *22nd Annual Computer Measurement Group Conference*, Dec 1996.

[30] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *22nd International Symposium on Computer Architecture*, June 1995.

[31] H. Wang, P. Wang, R. D. Weldon, S. Ettinger, H. Saito, M. Girkar, S. Liao, and J. Shen. Speculative precomputation: exploring use of multithreading technology for latency. In *Intel Technology Journal, Volume 6, Issue on Hyper-threading*, February 2002.

[32] P. Wang, H. Wang, J. Collins, E. Grochowski, R. Kling, and J. Shen. Memory latency-tolerance approaches for Itanium processors: out-of-order execution vs. speculative precomputation. In *8th International Symposium on High Performance Computer Architecture*, Feb 2002.

[33] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *28th International Symposium on Computer Architecture*, July 2001.