To CMP or not to CMP: Analyzing Packet Classification on Modern and Traditional Parallel Architectures

Randy Smith
Dan Gibson
Shijin Kong

Technical Report #1652

February 2009

To CMP or not to CMP: Analyzing Packet Classification on Modern and Traditional Parallel Architectures

Randy Smith   Dan Gibson   Shijin Kong
Department of Computer Sciences
University of Wisconsin
1210 W. Dayton Street
Madison, WI, U.S.A., 53706
{smithr,gibson,krobin}@cs.wisc.edu

Abstract

Packet classification is a central component of modern network functionality, yet satisfactory memory usage and overall performance remains an elusive challenge at the highest speeds. The recent emergence of chip multiprocessors and other low-cost, highly parallel processing hardware provides a promising platform on which to realize increased classification performance. In this paper we analyze the performance of packet classification in the context of parallel, shared-memory architectures. We begin with two classic algorithms–Aggregated Bit Vector and HiCuts–and parallelize each of them multiple ways. We discuss the tradeoffs of different architectures in the context of these algorithms, and we evaluate the schemes on both chip multiprocessor (CMP) and symmetric multiprocessor (SMP) hardware. Our experiments show that for CMPs, resource-sharing replaces synchronization scaling as the primary speedup-limiting bottleneck. Further, while SMPs provide more processing power core-for-core, CMPs nevertheless provide the best overall performance when all available execution contexts are employed.

1 Introduction

Packet classification is a central component of modern network functionality and is used for a variety of services, including routing, firewalls, and quality of service. Even so, achieving satisfactory performance with reasonable memory usage remains an elusive challenge at the highest speeds, and new techniques continue to be proposed to advance the state of the art. In large part, this ongoing challenge arises due to the tradeoffs between memory usage and execution time imposed by the nature of packet classification itself, combined with the need for very fast routines that can keep up with ever-increasing network speeds.

The recent emergence of Chip Multiprocessors (CMPs) [2, 9] and other low-cost, highly parallel architectures provide a promising platform for realizing increased performance for packet classification and for other common tasks in the forwarding path. This naturally leads to the question: what are the consequences–both positive and negative–of performing classification on parallel architectures? Increased performance is the primary motivator, but in addition, some CMPs provide more attractive performance per unit power than their uniprocessor or traditional multiprocessor cousins [11]. In addition to a potential power advantage, the use of multiple processing elements on a single chip has significant performance per unit cost advantages over multi-chip systems and clusters of networked workstations [19]. On the other hand, migrating to parallel architectures is no simple feat, and the parallelization process gives rise to hidden pitfalls and other constraints that limit scalability and must be addressed. These include extracting sufficient parallelism out of existing algorithms, devising altogether new algorithms, controlling inter-processor communication overhead, and limiting synchronization bottlenecks.

This work details our experiences in parallelizing packet classification algorithms and evaluating them on various multiprocessor architectures. We start with two classic but different classification algorithms–

Aggregated Bit Vector (ABV) [1] and HiCuts [8]–and parallelize each of them in multiple ways. We discuss the tradeoffs inherent to different parallel architectures in the context of these algorithms, and we evaluate their behavior on both Chip Multiprocessor (CMP) and Symmetric Multiprocessor (SMP) hardware. We consider the costs and benefits of parallel classification along three different axes. The first axis is the degree of parallelization, determined by number of processors available on the host machine. The second axis captures the features of the parallel hardware platform itself, including variables such as synchronization and locking costs, processor topology (CMP or SMP), and memory hierarchy differences. Third is the classification algorithm used and its parallel variants.

Our results show that parallel classification performance depends strongly on many factors, including algorithm selection, hardware platform, and parallelization scheme. In general, we find that each of the axes above affects the performance of classification dramatically and non-orthogonally with respect to the other axes. More specifically, we find that for CMPs, resource-sharing replaces synchronization scaling as the primary speedup-limiting bottleneck. Further, while SMPs provide more processing power core-for-core, CMPs nevertheless provide the best overall performance when all available hardware execution contexts are employed. Finally, for intermediate numbers of allocated processors, performance strongly depends on the allocation order itself. In the best cases, hardware costs and constraints are mitigated by the parallelization scheme (and vice versa), and we observe near-linear performance increases as the degree of parallelization increases.

In summary, at the very highest speeds packet classification may continue to be performed using custom hardware and software solutions. Nonetheless, the computing industry's trend toward multi-core parallel processing is clear. In this paper, we take a step towards understanding the costs of moving packet classification to parallel architectures and determine what benefits we can reasonably expect to obtain. Our results suggest that parallel classification on modern CMP hardware opens the door for further performance gains.

The remainder of this paper is organized as follows. Section 2 defines the packet classification problem and gives the related work, and Section 3 describes the architectural aspects relevant to our study. In Section 4 we briefly describe the ABV and HiCuts algorithms, and in Section 5 we present a taxonomy of parallelization schemes for these algorithms. Section 6 contains our experimental results, and Section 7 concludes.

2 Background and Related Work

A packet classifier compares packets to a database of rules to determine the lowest-cost rule matching each packet. Packets are classified according to the values of specific fields in their headers, which typically include source and destination addresses, source and destination ports, and the protocol. Rules in the database contain values for each of the five fields, a label, and a cost. Entries in the fields of a rule may contain explicit values (such as a specific IP address), a prefix, a range of values (*e.g.* a port range of [1024:65535]), or a wildcard to indicate that the field's value in the packet header is of no interest in the rule. When the lowest-cost matching rule R is found, R logically affixes its label to the packet. In the case of routing, for example, the label specifies the route the classified packet should take on its next hop. In practice, a rule's cost is determined by its index into the table of rules, with lower cost rules occurring first.

Conceptually, a classifier compares a packet's header against each rule in sequence until a match is found. But practically this approach is unacceptably slow, since rule databases often contain hundreds to thousands of entries, and classification must be performed at wire-speed to avoid creating a bottleneck. Further constraining the space of solutions, Lakshman and Stiliadis [10] showed that packet classification is an instance of a space-time trade-off [3], requiring either $O(log\ n^{k-1})$ time or $O(n^k)$ space for n rules and k header fields. Thus, proposals for efficient packet classification center on novel ways for achieving acceptable *trade-offs* between time and space. Rather than describe the myriad techniques in detail, we refer the reader to available summaries [7, 15, 17] and briefly describe here the higher-level qualitative and structural differences between them.

Most classification techniques can be divided along two axes: software vs. hardware (TCAM), and decomposition-based vs. heuristic approaches. Along the software–hardware axis, software-based classification is the most flexible and enables more sophistication and complexity at the cost of slower execution time. TCAM-based approaches, on the other hand, use specialized circuitry to classify a packet against all rules in parallel. TCAMs are fast but require large amounts of power and physical space and are expensive compared to commodity hardware. In addition, they have limited matching capabilities, since port ranges cannot be directly matched and must first be converted to lists of prefixes requiring extra capacity. A detailed summary of the trade-offs involved between software and TCAM-based approaches can be found in [6, 15].

Along the decomposition vs. heuristic axis we can see the influence of the space-time trade-off. Decomposition techniques perform classification by subdividing the problem into smaller, easier-to-compute components and combining the partial results in subsequent phases of the computation. From a parallelization perspective, these approaches are comparatively straightforward to parallelize since the work performed by each subtask is relatively constant and easy to determine *a priori*. In contrast, heuristic approaches use decision trees [8, 14, 18], prefilters [6], and other techniques to find the lowest-cost matching rule as quickly as possible. For these techniques, the amount of work per packet varies depending on the contents of the packet's fields and the structure of the heuristic. Parallelizing these approaches is more difficult, since balancing the work of each processor is not as straightforward.

In relation to these paradigms, our work investigates software-based techniques and rests on the assumption that multi-core processors with large numbers of cheap, lightweight cores may become viable platforms upon which classification is executed. For our experiments we parallelize and evaluate both a decomposition-based algorithm (Aggregrated Bit Vector [1]) and a heuristic decision-tree based approach (HiCuts [8]). We describe both of these techniques in Section 4.

With regard to network processing architectures, Yi and Gaudiot [20] have recently explored the applicability of extending network processing architectures with Simultaneous Multi-Threaded (SMT) capabilities. They evaluated all nine applications (none include packet classification) in the NetBench suite [13], concluding that simultaneously executing applications from different layers in the protocol stack gave the greatest performance improvement. In contrast, we study packet classification in detail and examine the effect that processor architecture has on parallelization scheme and vice-versa. In earlier work along these lines, Crowley *et al.* [5] found that SMT processors outperformed CMPs, Fine-Grained Multi-Threaded processors (FGMT) and other architectures. While there are many differences, our results are roughly consistent with theirs.

Finally, recent work has looked at approaches to parallelizing packet classification itself. In [21], Zheng *et. al.* propose a TCAM-based technique that uses chip level parallelism to exploit increased performance. In [22], the authors parallelize by *partitioning the rule sets* and running multiple decision tree-based classifiers in parallel. Each tree covers a distinct subset of the rules. The purpose of our work is not to propose new parallel classification algorithms *per se*, although we do parallelize some classic algorithms. Instead, we focus on the costs and behaviors of parallelization with regard to the architectures they are run on in an attempt to better guide current and future parallelization efforts.

3 Architecture Preliminaries

We study packet classification algorithms in the context of two shared-memory architectures. *Symmetric Multiprocessors* (SMPs) employ multiple processing *chips* to provide a parallel execution environment. Each such chip is allocated a single processor and a large, private cache. In contrast, *Chip Multiprocessors* (CMPs) integrate multiple processors onto a single chip, commonly sharing cache resources as well. While SMPs and CMPs present identical programming interfaces, applications will see very different performance characteristics due to architectural differences.

SMPs devote the vast majority of the area of a particular chip to a single processor and its cache hierarchy. The SMPs in this study provide a great deal of resources to a single execution context (*thread*), including large unshared caches that greatly reduce average memory access latency. The consequence of this chip allocation is that inter-thread (and, in this case, inter-processor) communication must cross at least two sets of pins, which incurs latency comparable to a cache miss or longer. While this operation is logically transparent to the programmer by virtue of cache coherence, it is costly, especially if it occurs with regularity.

In contrast, inter-thread communication is much less costly in a CMP environment. Communication between two threads residing on the same chip does not require any off-chip data movement and is therefore fast. The threads in our CMPs share a single unified (L2) cache on-chip which reduce communication time to only a few processor cycles. As a result, applications that are sensitive to inter-thread communication time are well-suited to a CMP environment. Unlike the SMP, however, the CMP must share some on-chip resources with other threads. Notably, additional cores and thread-private caches consume chip area and effectively reduce the performance of an individual thread in isolation. Ideally, this reduction in single-thread performance can be ameliorated by use of additional threads, though many applications do not trivially exhibit abundant thread-level parallelism.

The CMPs in our study also employ *multithreaded processors*, which greatly increase the number of available hardware threads by time-sharing each processing core at a fine granularity, but has the tendency to further reduce single-threaded performance. Specifically, as increasingly more thread contexts are used, the performance of each individual thread suffers as the thread's timeshare of its CPU is further reduced.

```
Thread 1             Thread 2
-----------------------------------
L.lock();            L.lock();
A = 27;              S = A + B;
B = 19;              L.unlock();
L.unlock();          do_work(W);
do_work(W);
```

Figure 1: Simple synchronized code segment

To illustrate these costs, consider the code sequences in Figure 1, depicting a synchronized critical section. Thread 1 writes two cache lines (A and B) which are protected by lock L. Thread 1 then performs W units of independent work requiring no communication. At a later time, Thread 2 acquires lock L and then reads cache lines A and B. Suppose Threads 1 and 2 execute in an SMP environment (*i.e.* on Chip 1 and Chip 2, respectively). Data requests for L, A, and B must all travel from Chip 2 to Chip 1 in the common case, and the data must then return to Chip 2. Each of these round-trip accesses may require hundreds of processor cycles to complete, incurring a large execution latency for a short synchronized section of code. However, if Threads 1 and 2 execute on the same chip, then *no* inter-chip traffic occurs, resulting in much faster execution. Since the threads in question share a common on-chip cache, communication time is reduced from comparable to a cache miss to comparable to a cache hit.

The whole of the execution does not consist of communication alone, however. For SMPs, work W is typically performed faster overall than on CMPs due to the superior single-thread performance of the SMPs. Hence, the *frequency of communication* becomes a vital indicator of SMP- or CMP-affinity. Simply put, when communication dominates CMP-based systems will tend to outperform SMPs. Conversely, SMPs tend to outperform CMPs when computation.

4 Classification Algorithms

We parallelize two distinct types of packet classification algorithms: the Aggregated Bit Vector [1] algorithm, which is decomposition-based, and HiCuts [8], which uses decision trees. We briefly describe each algorithm to provide context for the parallelization we describe later.

4.1 Aggregated Bit Vector

The Aggregated Bit Vector (ABV) scheme is itself an extension of the Lucent Bit Vector approach (LBV) [10]. LBV is a classic divide-and-conquer algorithm that subdivides the classification problem, solves the subtasks, and combines the intermediate results to compute a final answer. To perform the decomposition, LBV partitions the rule database by projecting along each of the k fields in the rules. In each of the k resulting sets, LBV then constructs a trie from the binary representation of the values in the rules for the given field. Values in a field that are ranges are converted to prefixes prior to trie construction. Each node in the tries contains a bitmap whose length is equal to the number of rules. For field j, bit i is set in the bitmap at a node N of trie j if the path from the root of the trie to N matches the prefix in field j of rule i. Altogether, the bitmap at N denotes the set of rules that match the prefix traced by the path to N.

When a packet arrives for classification, each of its k header fields is classified independently by following the associated tries as far as possible, resulting in k bitmaps which give the set of rules that match each field independently. To combine these subresults, LBV then intersects the k bitmaps and looks for the set bit whose rule has the lowest cost. If rules are ordered according to cost, the algorithm needs only to find the first bit that is set in the intersected bitmap.

The ABV algorithm extends LBV by drawing on the observation that the intersected bitmap is large and sparse, leading to a large number of unnecessary memory accesses and wasted cycles when looking for a set bit. In the extension, a small aggregate bitmap is also associated with each trie node that summarizes the regular bitmap. If A is the aggregate size, a bit i is set in the aggregate bitmap if there is at least one bit in the range $[i \cdot A, (i+1) \cdot A)$ that is set in the regular bitmap. During classification, the aggregated bitmaps are intersected and examined first. When a set bit is observed, the corresponding bits in the intersected regular bitmap are then examined. ABV reduces the number of bits to be examined from $|R|$ (the number of rules) to $|R|/A$, which can lead to large decreases in memory accesses for large values of A. Further improvements are possible by extending the aggregation beyond a single level to create a multi-level hierarchy of aggregate bitmaps.

4.2 Hierarchical Intelligent Cuttings (HiCuts)

Hierarchical Intelligent Cuttings (HiCuts) [8] is a heuristic, decision-tree [4] based procedure for performing classification. In this technique, nodes in a decision tree serve as filters that successively reduce the number of matching rules until a small enough number of rules remains for which linear search is feasible. Geometrically, rules can be viewed as points in a k-dimensional hypercube, where as before k is the number of header fields involved in the classification. Nodes in the decision tree correspond to planes that recursively subdivide the hypercube into smaller subcubes that classify the rules they contain. The goal of the decision tree problem is to provide a good partition of the rules while simultaneously performing as few cuts as possible (thus reducing the height of the tree). Determining the best possible decision tree is NP-Complete [17], so heuristics with reasonable performance are used to approximate an exact solution.

HiCuts uses heuristics that seek to balance the space and time costs and are based on providing the best balance based on locally optimal criteria. Large trees are fast but require large amounts of storage space, whereas small trees can use very little space but are slow since they result in nodes with large number of rules to be linearly matched. Tunable heuristics guide the construction of tree to some acceptable footprint between either extreme. Once the decision tree is constructed, performing classification is simple. A packet is classified by traversing the tree, evaluating the nodes' predicates in succession and following the appropriate child pointers. When a leaf node is reached, the classifier linearly scans the rules contained in the node and affixes the label of the lowest cost rule to the packet.

5 Parallel Classification: Techniques and Analysis

Packet Classification is computation-bound from a high performance linespeed networking perspective, but it is lightweight in comparison to many traditional parallel problems such as large scale scientific simulations. Communication overhead can be relatively, and understanding inter-thread communication cost is key to achieving high-performance packet classification via parallel software. We considered and implemented many parallelization schemes for the ABV and HiCuts algorithms, including data parallel, control parallel, and pipelined parallel approaches. Each of these schemes optimizes a different trade-off between various parallelization overheads. We describe each approach below as well as specific implementation issues.

5.1 Data Parallel Approaches

The simple data parallel approach is analogous to the *Single Instruction Multiple Data (SIMD)* execution model. In this approach, the classification algorithm is fully replicated onto each processor, and packet headers are multiplexed to an arbitrary processor for parallel classification. This method is straightforward and scales reasonably well, as virtually no inter-thread communication is required when classifying two separate packets. Initialization tasks that construct data structures are performed prior to the replication, and a single copy of these and other read-only data structures is shared among all instances of the algorithm.[1]

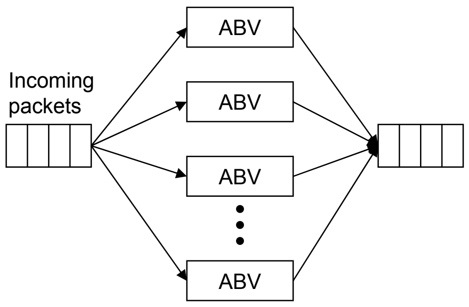

Figure 2: Data parallel Aggregated Bit Vector. Each processor executes the complete algorithm.

Figure 3 illustrates the data parallel approach for the ABV algorithm; the HiCuts data-parallel implementation is structured identically. A shared queue holds packets that have been received from the network and are awaiting classification, and a mutual exclusion lock guards the queue and arbitrates access to it among the competing processors. As with all our parallelizations, this lock simulates the synchronization that must occur when retrieving a packet from the network interface. On each processor, the classification cycle consists of three steps: busy-waiting to acquire the lock, retrieving a header from the queue (more generally, retrieving a set of N headers), and classifying and outputing the header (or N headers).

There are two primary advantages to data parallel schemes. The first is simplicity: parallelizing the algorithm requires only the addition of a lock and some logic at the entry point of the algorithm to synchronize the queue. The steps involved in the classification algorithms themselves remain intact; no surgery is required to extract the parallelism out of an algorithm. Secondly, this approach minimizes the dependencies between processors, which is not true of the other parallelization schemes we consider. Here, processors classify packets independently and do not need to execute in tight synchrony. Each processor can classify packets as fast as its capabilities allow, without regard to the speed or duty cycle of other processors.

The primary disadvantage of this approach is the use of a single lock guarding the shared queue. As characterized by Mellor-Crummy and Scott [12], this construction limits scalability for even small numbers of processors, especially on traditional parallel architectures such as SMPs that have high communication costs (Figure 12). Only one processor can hold the lock at a time, and acquiring the lock becomes a chokepoint as the number of processors increases. As we show later, our experimental results are generally consistent with those described in [12], although the bottleneck is mitigated somewhat in CMPs, where locking costs are cheaper.

There are many variants to the data parallel approach, including techniques that subdivide the computation tasks into smaller components and summarily optimize scheduling of subtasks (*e.g.*, a master thread or use of multiple thread-private queues). These techniques tend to increase the communication time overhead in addition to having the same locking problems as above.

5.2 Control Parallel Approaches

In contrast to data-parallelism, *control parallelism* subdivides an *algorithm* into multiple subparts that can be executed in parallel. Control-parallel approaches work well on algorithms that repeat the same tasks for different data values. For subparts that cannot be parallelized, such as bitmap intersection in the ABV algorithm, all processors performing tasks leading up to the serial subpart must communicate their results

[1]On most machines, no inter-thread communication is necessary when threads share cacheable data in a read-only manner.

to a designated processor. When a designated processor has received all the partial results, it executes the algorithm subpart serially until a parallelizable subpart is again reached. Thus a control-parallelized algorithm is often characterized by alternating parallel and serial components.

At first, control parallelization seems to be a good fit for the decomposition-based ABV algorithm. Trie traversals over header fields offer independent operations can be performed on distinct processors, and analysis of the aggregate bitmaps and rule bitmaps can be allocated among several processors as well. Figure 4 depicts a control parallel scheme for the ABV algorithm that leverages two parallel components, each followed by a serial component. The first component parallelizes trie traversal so that each header field's best matching prefix is identified on a distinct processor. Next, the aggregation bitmaps from each trie are intersected together in a serial operation. In the second parallel phase, the intersected bitmap is divided among the available processors and aggregation is performed. Finally, a single processor selects the lowest cost rule from each of the partial results produced.

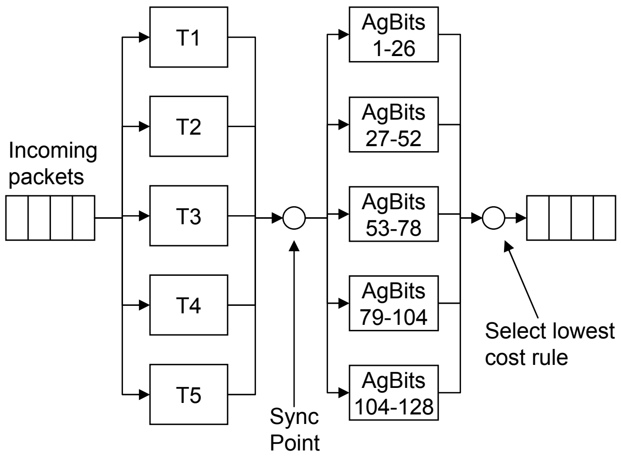

Figure 3: Control-parallel Aggregated Bit Vector.

Unfortunately, this parallelization has serious shortcomings. First, there is a heavy synchronization cost, as tight synchrony is required at all serialization points following the parallel phases, and barriers must be employed to ensure that serial phases do not execute until all dependencies have communicated their results. Time is lost as processors idle while waiting for a previous processor to reach the synchronization point and communicate its results. Second, the parallelization does not scale gracefully to more than five processors, since this is the limit of the available parallelization in the first phase of the ABV algorithm. Combined together, these two costs yield an unworkable solution, and initial results demonstrated that a performance *slowdown* occurs as the number of processors is increased. Further, the HiCuts algorithm admits no obvious amenity to such parallelization, since each node in the decision tree is dependent on its parent.

From another perspective, the poor behavior can be understood by considering the communication costs involved. For the ABV algorithm, our experiments have shown that, on average, 12-14 nodes are accessed during trie traversal, most of which reside in the cache, so that the computation time for the first parallelizable subtasks is fairly small compared to the aggregation steps. As a result, the communication costs are comparatively high, but these do not contribute to the algorithm and count as overhead. Control parallelization works well for tasks that have large, balanced computation times, unlike the disparity between ABV's trie traversal and aggregation steps. Thus, we conclude that control-parallel schemes are unsuitable for packet classification.

5.3 Pipelined Parallelism

Pipelined techniques are a special case of control parallelism restricted to an assembly-line model for data movement. Here, the classification task is partitioned into multiple stages; processors are connected together in sequence, and each processor is assigned a specific stage to execute. Small queues between each processor decouple the read and write operations of adjacent processors in the pipe and allow for a limited amount of buffering. During operation, each processor waits for data on its input queue, reads in the data when it arrives, executes its assigned stage(s), and places the intermediate results in its output queue for the next processor in the chain.

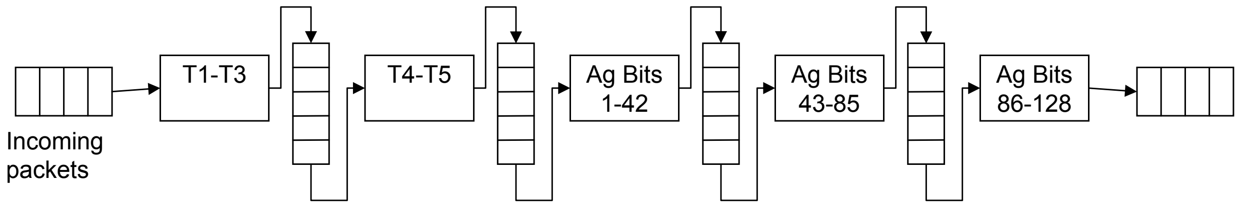

Figure 4: Pipelined Aggregated Bit Vector algorithm. Each processor completes a portion of the task and hands off the intermediate results to the next processor.

The chief advantage of the pipelined approach is reduced lock contention. Each intermediate queue between processors has a lock that regulates access to the queue's contents. Since these queues are shared by only two processors, contention for the lock is considerably reduced compared to data-parallel approaches. In principle, throughput can be higher than a data parallel approach since lock manipulation and wait time can be much lower in a pipelined implementation. The chief obstacle to this approach is that overall throughput is throttled by the worst-performing task in the pipeline. Thus, in practice, increased throughput depends strongly on a balanced workload among all the processors in the pipeline.

Figure 4 shows a pipelined parallelization of the ABV algorithm for five processors. A lock guards access to each intermediate queue. Stage 1 busy-waits for packets to be received from the network, and Stages 1 and 2 together perform the trie traversals. The remaining three stages perform the tasks of intersecting the bitmaps from the tries and finding the lowest cost matching rule. We scale the parallelization up to higher numbers of threads by spreading the tries among distinct processors and by dividing the bitmap evaluation step into smaller subranges. In our implementation, to avoid expensive (and unnecessary) copies of intermediate results, packet headers along with their intermediate results are maintained in globally visible memory, and pointers to their location in memory are communicated between processors.

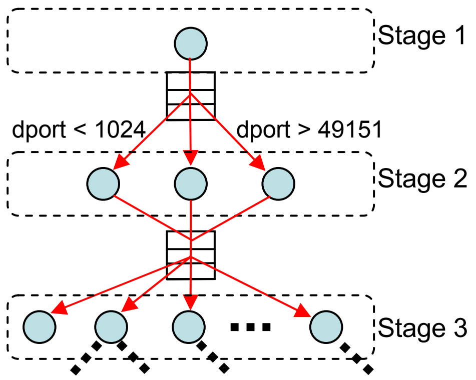

Figure 5: Pipelined HiCuts. Processors in the pipeline are assigned to distinct levels in the decision tree.

With regard to packet classification, our experience has been that decomposition-based techniques such as ABV are easy to parallelize since they can be decomposed into smaller tasks that are relatively deterministic in their execution time, leading to well-balanced pipelines. Designing a pipelined parallelization (or a more general control parallelization) for heuristic approaches is not as straightforward. The overarching requirement is to produce a well-balanced decomposition that can keep all processors utilized. However, heuristic decision-tree based algorithms such as HiCuts interleave evaluation of different fields at all levels of the tree, rendering ABV-style decompositions impossible. An alternative approach, and the one we employ, is to decompose the decision tree so that each level in the tree is assigned to a distinct processor. Figure 5 depicts this technique for the first three levels of a tree. In addition, when available we dedicate one or more processors to performing the linear traversals at the leaves. As above, a small queue with a lock sits between each level. When the processing at any level has been completed, the processor places the intermediate results along with a child node identifier on its output queue.

Server	Sun T2000 (CMP)	SunFire v880 (SMP)
Processors	8 UltraSPARC-T1	8 UltraSPARC-III+
Threads	32	8
Chips	1	8
L1-I	16 KB	32 KB
L1-D	8 KB	64 KB
L2	3 MB(on chip,shrd)	8 MB(off chip, prvt)
Interconnect	Crossbar	Shared Bus
Main Memory	16 GB	32 GB

Figure 6: Characteristics of CMP and SMP platforms. The Sun T2000 (Niagara) provides a CMP execution environment; the SunFire v880 is an SMP.

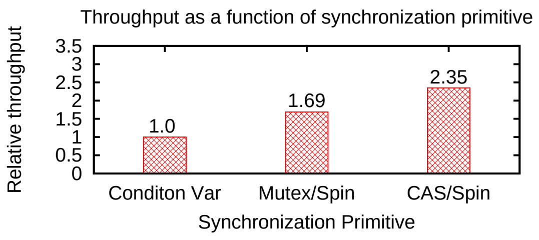

Figure 7: Relative classification throughput for various locking mechanisms. Using the CAS instruction with busy-waiting yields the highest throughput.

6 Experimental Results

We implemented data-parallel and pipelined parallelizations of both the ABV and HiCuts algorithms and evaluated them on CMP and SMP hardware, repeating all of our experiments for processing quantum sizes ranging from 1 to 100 packet headers. We briefly summarize our findings as follows:

- **Processor Allocation** On CMPs, resource-sharing is the primary speedup-limiting bottleneck. Further, thread/processor allocation order can significantly affect performance for intermediate numbers of allocated threads.
- **Synchronization Costs** Synchronization *scaling* costs did not produce significant bottlenecks on either platform, although SMP communication costs limited scalability and far outweighed CMP communication costs.
- **Parallelization Schemes** In part because of the above, Data Parallel approaches achieve close to ideal speedups, whereas workflow imbalances restrict pipeline performance.
- **Performance** Core for core, SMPs provide superior performance. When all available processor resources are used, though, CMPs yield the best overall performance.

6.1 Execution Environment

Figure 6 provides the details of the CMP and SMP platforms in our study. We used a Sun T2000 server (Niagara) for our CMP environment, which has 8 cores and 4 hardware threads per core for a total of 32 independent hardware contexts (threads). We used a SunFire v880 for the SMP platform containing 8 chips for a total of 8 hardware contexts. For some experiments, we also employed an older Sun E6000 16 processor SMP system as a comparison point. All experiments were performed on idle machines and repeated twice with the best result retained. We use the term *quantum* to refer to the number of packets processed between locking events. Unless otherwise noted, we use a quantum of 1. For our tests, we used synthetic classifiers and traces produced with the Classbench [16] framework.

One implicit degree of freedom in each of the algorithms we studied is the choice of synchronization primitive. We considered three distinct synchronization mechanisms: condition variables, pthread mutexes using busy-wait loops, and the atomic *compare-and-swap (CAS)* SPARC instruction. The relative throughput achieved using each of these mechanisms is given in Figure 7. Condition variables are the most expensive since they communicate synchronization via signals relayed through the operating system. Using busy-wait loops with pthreads eliminates much of the OS-signalling overhead and gives a 70% relative improvement. Finally, using the CAS instruction with busy-wait loops further reduced lock acquisition time and yielded 135% improvement over condition variables. Consequently, all our experiments use an inline CAS-based macro for synchronization in lieu of condition variables or library-provided locking.

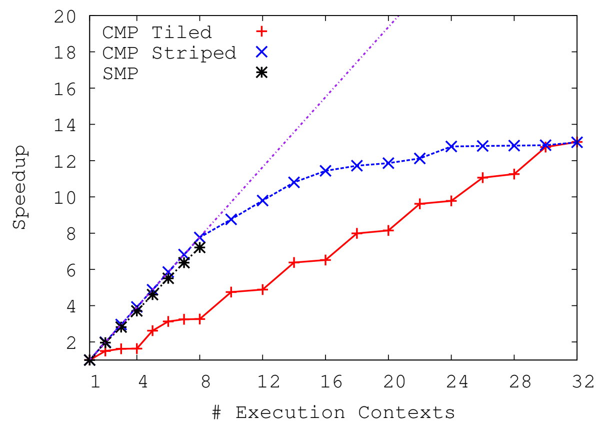

Figure 8: Data Parallel ABV

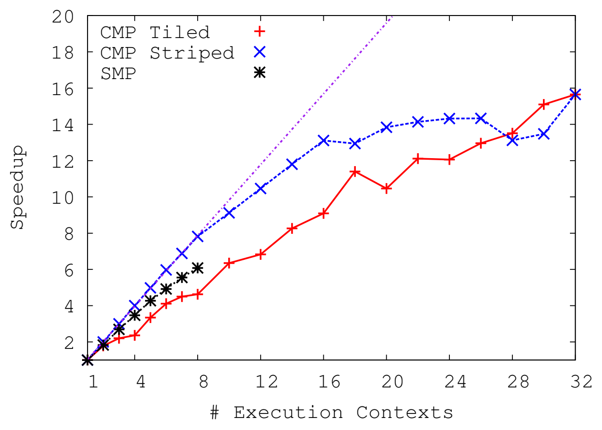

Figure 9: Data Parallel HiCuts

Parallel. Scheme	Algorithm	Platform	# Performance (pps)			Speedup	
			1	8	32	8	32
Data Parallel	ABV	CMP (Striped)	50,004	388,115	651,879	7.76×	13.03×
		SMP	51,933	374,226	–	7.21×	–
	HiCut	CMP (Striped)	113,523	888,203	1,777,019	7.82×	12.67×
		SMP	211,444	1,283,264	–	6.07×	–
Pipe Parallel	ABV	CMP (Striped)	43,448	200,460	195,006	4.61×	4.49×
		SMP	45,990	150,463	–	3.27×	–
	HiCut	CMP (Striped)	100,908	149,838	383,024	1.60×	2.76×
		SMP	166,088	28,900	–	0.17×	–

Table 1: Raw performance (in packets per second) and speedups for 1, 8, and 32 concurrent threads.

6.2 Data Parallel Results

Figures 8 and 9 depict the speedup of the data-parallel ABV and HiCuts algorithms, respectively. The dashed line shows the ideal speedup as a reference. On the CMP, performance increases for intermediate thread counts (*i.e.* less than 32) strongly depend on the order in which threads are allocated. A *tiled* allocation uses all threads in the current core before allocating a thread from another core. The *striped* allocation scheme, on the other hand, balances thread allocation among all cores by allocating threads so that they are spread across all cores as evenly as possible. Since threads on the same core share resources, striped allocations that do not share resources outperform their tiled counterparts for intermediate thread counts. As expected, the distinct allocations converge at full utilization.

For both algorithms, CMP speedup is near the ideal for eight or fewer striped threads (each on a distinct core) and gradually falls off as threads on the same core compete for processor resources and become increasingly constrained. For CPU-intensive workloads, CMP scalability is limited by CPU performance saturation rather than communication costs. Note that on the SMP platform, scalability falls from the ideal in both algorithms. We attribute this to the relatively increased communication cost of the SMP architecture. Further, since HiCuts is a more efficient algorithm than ABV, the ratio of communication to computation is higher and the fall-off from the ideal is larger in Figure 9 than in Figure 8. Table 1 shows the raw performance numbers in packets per second and the resulting speedups.

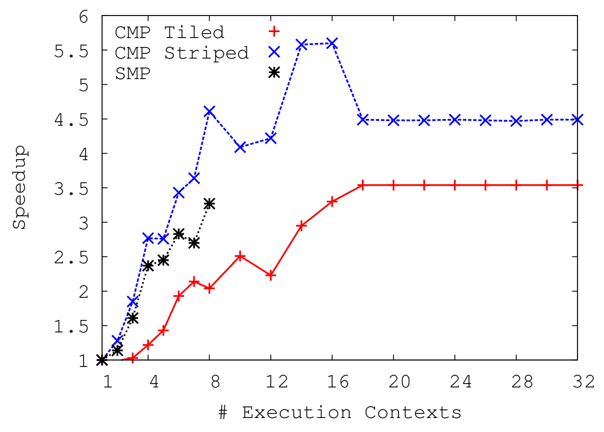

Figure 10: Pipelined ABV

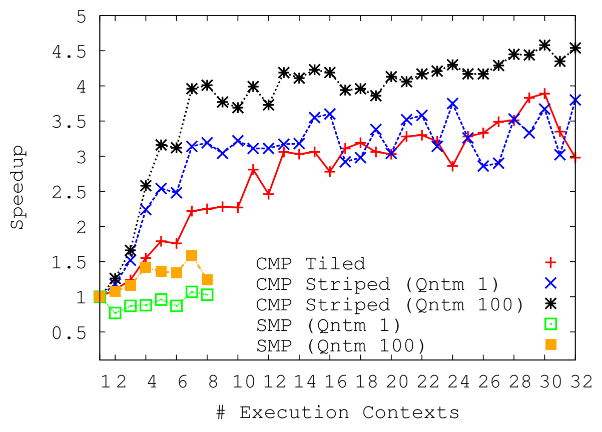

Figure 11: Pipelined HiCut

6.3 Pipelined Parallel Results

Pipelined parallelizations exhibit characteristically different behavior as shown in Figures 10 and 11 (Figure 11 also shows behavior for larger quantum sizes as discussed in the next section). Most significantly, the observed speedup is quite modest compared to data parallel approaches, due to both increased communication costs and processor load-balancing issues. Increasing the number of threads increases the number of synchronization points that must be traversed; even though only two threads contend for a given lock, the frequency of communication is larger and thus communication costs contribute larger overhead. Subdividing the algorithm into small enough pieces so that it can be spread among the available processors further increases relative communication costs, although this can be addressed somewhat by increasing the processing quanta. At some point, tasks are broken down into their smallest “atomic units”, which do not necessarily have uniform processing requirements. This imbalance in the processing loads of the pipeline constrains overall performance to that of the slowest thread. Such imbalance manifests itself partially as non-monotonic architecture-specific “sweet spots” shown in the figures.

Second, CMP performance does not converge at 32 threads as is the case for the data parallel results. This behavior stems from the variable synchronization costs that occur between pipeline stages. Stages that reside on different cores will have different communication costs than those that reside on the same core. These costs lead to significant speedup differences even when all processors are utilized since different paths between the threads and cores are followed.

Finally, SMP behavior varies widely between the two algorithms. Pipelined ABV loosely follows the pipelined CMP trend, but no significant speedup is observed with pipelined HiCuts. This stark contrast between SMP and CMP behavior comes (again) from the relatively heavy synchronization costs for SMPs compared to CMPs as well as the shared on-chip L2 cache on the CMP. When combined with the reduced computation requirements of HiCuts over ABV, the resulting high communication to computation ratio of HiCuts on SMPs effectively constraints any speedups.

6.4 Varying the Quantum Size

As alluded to earlier, varying the quantum size–the number of packets processed between synchronization events–lowers the communication to computation ratio and can positively affect performance. We examined the impact of this by executing the algorithms as the quantum size varies from 1 packet to 100 packets per synchronization event. Table 2 summarizes these results. In the table, the second column contains the best performance with quantum 1 using the maximum number of threads available to the platform. The

Algorithm & Platform	Performance (Quantum 1)	Max Perf.	Best Quantum	% Improv.
ABV DP CMP	651,879	743,559	70	14.1%
ABV DP SMP	374,226	427,908	60	14.3%
ABV PIPE CMP	195,006	232,875	50	19.4%
ABV PIPE SMP	150,463	198,303	40	31.8%
HiCuts DP CMP	1,777,019	2,024,021	70	13.9%
HiCuts DP SMP	1,283,264	1,622,839	40	26.5%
HiCuts PIPE CMP	383,024	476,740	100	24.5%
HiCuts PIPE SMP	188,840	240,248	100	27.2%

Table 2: Performance improvements obtained when the number of processed headers per synchronization event (quantum) is increased.

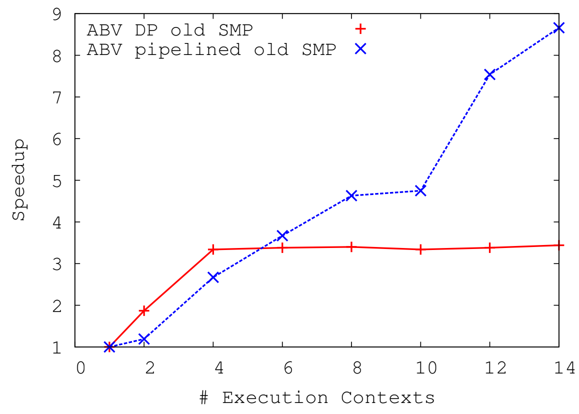

Figure 12: Pipelined ABV vs. Data Parallel ABV on an older E6000 SMP. Lock contention restricts data parallel scalability, whereas the pipelined approach readily scales.

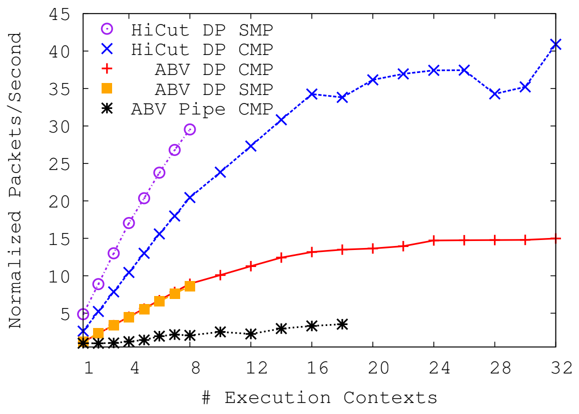

Figure 13: Comparing the relative performance of ABV and HiCuts using various parallelizations.

third column contains the best overall performance we observed, using the same number of threads, as the quantum is varied. The fourth column contains the quantum size at which the best performance was observed, and the rightmost column shows the percentage improvement.

Overall, the best improvements come from the HiCuts algorithm. Relatively high communication overhead is a natural consequence of the increased efficiency of that algorithm; increasing the quantum size provides an easy way to reduce the overhead. Somewhat surprisingly, performance does not monotonically rise with continued quantum size increases, as indicated by quantum sizes that are less than 100. As the quantum size grows, the amount of buffering (and copying) between processors increases. Further, pipeline imbalances can lead to full buffers that may induce blocking during processing. Thus, the best quantum size is not the largest possible, but rather the size that minimizes the total throughput delay.

Despite these improvements, the trends in Figure 11 put them in perspective: increased quantum size can improve performance, but it does not fundamentally tip the scales in favor of one algorithm or architecture over another. Further, it is not clear whether the delay introduced by batch-processing up to 100 packets at a time falls within acceptable bounds. Thus, varying the quantum size is better employed to enhance performance rather than as a selection factor.

6.5 Discussion

Classification Performance. Our experiments have focused on the scalability of parallel classification algorithms rather than the raw performance as a means for gauging the potential of CMP and SMP architectures and also to ensure an even evaluation. Looking specifically at performance, Figure 13 summarizes the results for many configurations normalized for display purposes to the slowest configuration (ABV pipelined). Our results confirm others [17] that indicate HiCuts outperforms ABV. Further, SMPs provide superior performance thread-for-thread. At eight threads, SMP performance dominates CMP performance on the target platforms. However, we find that when all computational resources are employed, data-parallel HiCuts CMP provides the best overall performance even in the presence of resource-sharing.

Parallelization techniques. Given that data parallel approaches consistently outperform their pipelined counterparts, one may ask–what value are pipelined approaches? Historically, synchronization scaling [12] has been a bottleneck to obtaining speedups for even modest numbers of processors. Figure 12 shows a comparison of data parallel and pipelined ABV parallelizations on an older Sun E6000 16-processor SMP. Here, synchronization scaling becomes a bottleneck with as few as four processors, and pipelined approaches are needed to obtain linear speedups. In contrast, our current results show that for this problem, the engineering constants impeding synchronization scaling have changed, and that data parallel approaches are acceptable for larger numbers of processors.

7 Conclusion

This work details our experiences in parallelizing packet classification algorithms and evaluating them on Symmetric Multiprocessor (SMP) and Chip Multiprocessor (CMP) architectures. We find that although the CMP architecture provides the best overall performance, there are many significant factors that influence the overall behavior. First, for smaller number of execution contexts, SMPs may provide the best performance. In addition, resource-sharing replaces synchronization scaling as a speedup-limiting bottleneck, and CMP performance is sensitive to processor allocation order. We find further that extracting sufficient parallelism from the classification algorithms is challenging, and that data-parallel algorithms provide the easiest path to improved performance.

CMPs are a promising environment for high-performance parallel packet classification in software. While the highest-performing links will likely still be classified with hardware methods, CMPs offer attractive performance per watt and performance per unit cost though thread-level parallelism. The CMP's natural tolerance for inter-thread communication optimizes the necessarily small computation time per packet, and does not mandate the processing of many packets at a time to provide scalability.

References

[1] Florin Baboescu and George Varghese. Scalable packet classification. In *SIGCOMM '01*, pages 199–210, 2001.

[2] Luiz A. Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 282–293, 2000.

[3] Paul Beame and Erik Vee. University of washington time-space tradeoffs site. Available at `http://www.cs.washington.edu/homes/ beame/projects/timespace.html`.

[4] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[5] Patrick Crowley, Marc E. Fiuczynski, Jean-Loup Baer, and Brian N. Bershad. Characterizing processor architectures for programmable network interfaces. In *ICS*, pages 54–65, 2000.

[6] Qunfeng Dong, Suman Banerjee, Jia Wang, and Dheeraj Agrawal. Wire speed packet classification without tcams: a few more registers (and a bit of logic) are enough. In *SIGMETRICS*, pages 253–264, 2007.

[7] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, March 2001.

[8] Pankaj Gupta and Nick McKeown. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, 2000.

[9] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. 25(2):29–35, Mar/Apr 2005.

[10] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of the ACM SIGCOMM*, pages 203–214, New York, NY, USA, 1998. ACM Press.

[11] James Laudon. Performance/watt: The new server focus. *Computer Architecture News (CAN)*, 33(4), 2005.

[12] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. In *ACM Trans. on Computer Systems. February 1991, pp. 21-65.*, 1991.

[13] Gokhan Memik, William H. Mangione-Smith, and Wendong Hu. Netbench: a benchmarking suite for network processors. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 39–42, 2001.

[14] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *SIGCOMM*, 2003.

[15] David E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv.*, 37(3):238–275, 2005.

[16] David E. Taylor and Jonathan S. Turner. Classbench: a packet classification benchmark. In *INFOCOM*, pages 2068–2079, 2005.

[17] George Varghese. *Network Algorithmics, An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann, 2005.

[18] Thomas Y. C. Woo. A modular approach to packet classification: Algorithms and results. In *INFOCOM*, pages 1213–1222, 2000.

[19] David A. Wood and Mark D. Hill. Cost-effective parallel computing. *IEEE Computer*, 28(2):69–72, 1995.

[20] Kyueun Yi and Jean-Luc Gaudiot. Architectural support for network applications on simultaneous multithreading processors. In *IPDPS*, pages 1–10, 2007.

[21] Kai Zheng, Hao Che, Zhijun Wang, Bin Liu, and Xin Zhang. Dppc-re: Tcam-based distributed parallel packet classification with range encoding. *IEEE Trans. Computers*, 55(8):947–961, 2006.

[22] Kai Zheng, Zhiyong Liang, and Yi Ge. Parallel packet classification via policy table pre-partitioning. In *Proceedings of GLOBECOM '05: Global Telecommunications Conference 2006*, dec 2005.