A Co-Processor Approach for Accelerating Data-Structure Intensive Algorithms

Jason Loew Jesse Elwell Dmitry Ponomarev Patrick H. Madden
SUNY Binghamton Computer Science Department

***Abstract*— Many important software applications are dominated by non-trivial serial components: Amdahl's Law places a hard upper bound on possible speedup that can be achieved for these applications. In this paper, we propose an integrated software/hardware approach for accelerating hard serial bottlenecks in data structure heavy algorithms. The key idea is to overlap the processing of the main algorithmic functions and the data structure related operations.**

We describe the language, compiler, ISA and architectural support for such data structure co-processing (DSCP), and define a clean interface between the software and the hardware. We perform extensive simulations using the popular C++ STL container classes, as well as a detailed implementation of our approach for Dijkstra's single-source shortest path algorithm. We find potential for improvements that are well beyond what can be achieved with more conventional parallel computation methods.

I. Introduction

The microprocessor design industry has recently undergone a fundamental shift from complex single-core architectures to multi-core systems with simpler individual cores. These emerging multicore and many-core architectures provide unprecedented opportunities for parallelism and can achieve high overall chip throughput for highly-parallelizable workloads or for workloads comprised of a large number of independent programs. However, it is much more challenging to sustain performance improvements of single-threaded applications (or sequential parts of multi-threaded applications) in these environments. Since the core clock frequency has remained relatively constant for the last few years and is not expected to increase due to power and thermal considerations, future significant performance improvements can only come from microarchitectural innovations. However, if a multicore system is constructed by simply replicating existing cores, then performance of sequential code will, in fact, degrade compared to performance on a single-core machine. This is due to the effects of last-level cache and memory sharing, especially as the number of cores per chip increases. The situation is further exacerbated when simpler cores are used.

Achieving high performance in executing sequential code is crucial because many important applications are strictly sequential in nature and they cannot directly make use of the parallelization opportunities afforded by the multicore hardware. Furthermore, parallel programs often have significant sequential portions, the performance of which determines the speedup achievable by these applications on parallel architectures. This is an instance of Amdahl's Law[2]. Compounding this situation is the fact that when a sequential portion of a parallel application is executed on one of the cores, the other cores dedicated to this application remain idle, thus decreasing the overall power-performance efficiency of the multicore chip [9].

In this paper, we focus on accelerating hard serial bottlenecks – sections of a program which are not trivially parallel. In particular, we concentrate on algorithms that perform extensive manipulations with data structures. These algorithms typically consist of the actual algorithmic activities and the related data structure management operations. The interface between an algorithm and its related data structure provides a "clean break" between two activities, allowing overlapping of work with minimal bookkeeping overhead. Consider, for example, the insertion of a value into a binary search tree. The insertion routine must traverse the tree to an appropriate location, revise pointers, and potentially rebalance the tree – the amount of computation required depends on the size of the tree, and a variety of other factors. From the perspective of the main algorithm, however, there is no need to wait for this insertion to complete – other processing may continue, as in the ideal situation, the entire internal structure of the tree is encapsulated by the data structure. This is illustrated in Figure I.

Obtaining parallelism by offloading portions of work is by no means new. A number of prior efforts considered such parallelization, these efforts are described in more detail in the next section. Our work is novel in that we focus on the interface between fine-grain data structure operations and algorithmic work. To the best of our knowledge, no prior work explored the concept of off-loading computations to separate processing engines at the level of data structure operations. We demonstrate through experiments that even simple operations on common data structures require hundreds or even thousands of processing cycles, even for the modest sizes of the underlying data structures. Finally, we demonsrate tha entire concept using the example of Dijkstra's shortest path algorithm.

Specifically, we propose an integrated hardware-software framework to dynamically off-load the data structure related processing to a separate processing unit (either a separate core within a multicore chip or a separate thread within a multithreaded processor, or a specialized core) and support its concurrent processing with the main algorithmic activities. We call this paradigm Data Structure Co-Processing (DSCP). DSCP requires modest programming language and compiler support to provide a clear communication interface between the activities of the main thread and the thread performing

978-1-4244-8935-0/10/$26.00 ©2010 IEEE

the data structure operations.

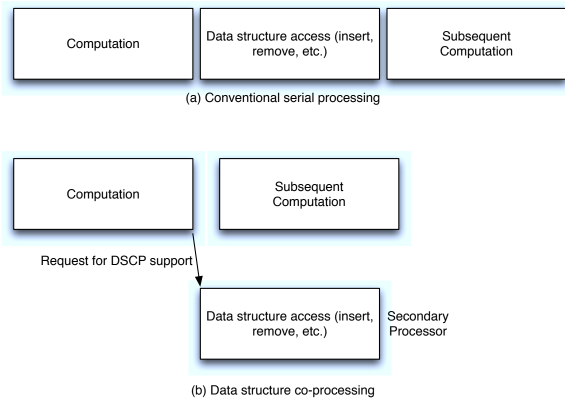

Fig. 1. The basic data structure co-processing approach: rather than executing data structure maintainence instructions on the main thread, a request is made to a secondary processor. Depending on the type of operation performed, it may be possible for the main thread to continue computations, rather than waiting for the data structure operation to complete.

II. Background

Several recent efforts explored the use of multiple cores or multiple thread contexts within a multithreaded processor for accelerating sequential single-threaded applications. Largely, these techniques can be categorized into four groups: asymmetric multi-core architectures [7], [9], [20], core fusion techniques [10], [11], [18], speculative multithreading and pre-execution [6], [16], [19], [22], [23], [17], and compiler/architecture techniques for extracting loop-level parallelism. We now describe the prior efforts in each of these groups in some detail.

Fusion of multiple CMP cores to collectively execute a single-threaded application has been a topic of recent interest. The work of [10] described the hardware support needed to seamlessly fuse multiple out-of-order cores and offered comprehensive performance evaluation of this technique. In this design, the cores perform collective fetch, decode, renaming, scheduling, execution and commitment of instructions. The technique is reminiscent of clustered microarchitecture designs, but the need for cross-core communication imposes larger overhead both in terms of performance and complexity. Other proposals [18] considered fusion of in-order cores and concluded that such a model has fundamental performance challenges and limitations. In addition, a mechanism to aggregate the individual cores of TRIPS distributed microarchitecture has been proposed in [11]. The TRIPS architecture requires special ISA and the compiler support. Unfortunately, simple aggregation of multiple cores only provides a larger and more powerful processing core, but it offers little help in addressing the problem of speeding up the inherently sequential operations with data structures that we consider in this paper. This is due to the very fine-grain nature of data structure related operations, and frequent interactions with the rest of the code. The core fusion approach does not address the root of the performance problem inherent in many algorithms - namely, that the maintenance of the priority queue and the operations performed with the extracted elements are serialized. Our proposal addresses this fundamental bottleneck.

Another well-researched avenue for improving single-thread performance is speculative multithreading [1], [6], [16], [19], [22], [23], [17], where helper threads are speculatively executed on the spare cores either to provide prefetching and branch prediction for the main thread [16], [22], [23] or execute relatively independent tasks further along the control flow of the main thread [6], [19]. If the decomposition into threads is done in a coarse-grain fashion, as was the case with most early works, then the benefits of this paradigm are constrained due to many inter-thread dependencies, especially in difficult-to-parallelize codes. In the most recent example of such approaches [6], the compiler decomposes the instructions from a single-threaded application across multiple cores in a fine-grain fashion, and the hardware includes special components to support this execution model. In either case, the speculative nature of these designs entails significant complexity for forking off threads, integrating the results and handling memory dependency violations among the threads, and recontructing original program order and creating checkpoints. In contrast, our proposal requires a very modest hardware support in the form of a 2-entry inter-core buffer and the overlapping of data structure operations with the main algorithmic activities (which use the results returned by the data structure) is accomplished in a non-speculative manner.

Asymmetric (or heterogeneous) multicore architectures represent one approach for efficiently executing performance-critical portions of the program [7], [9], [20]. In the approach of [20], selected critical sections execute on a large core of an assymmetric chip multiprocessor, at the request of a small core if necessary. Executing critical sections on the large core ensures that the lock and shared data always stays in the cache hierarchy of the larger core instead of constantly moving across cores. Several efforts proposed to hide the latency of critical sections by speculatively executing them concurrently with other instances of the same critical sections as long as they do not have data conflicts with each other [8], [14]. The critical sections used in parallel applications (which are the target of the work of [20]) are generally different from hard serial bottlenecks that we address in this paper.

Several techniques have been proposed for accelerating loops in sequential applications to exploit the normally hidden loop-level parallelism. In [21], a compiler-based approach is described. In [3], novel loop acceleration architecture and the dynamic algorithm for mapping loops onto the loop accelerators are presented and analyzed. These techniques are synergistic with the approach proposed in this paper.

The concept of active pages was proposed in [13] in an effort to create a model of computation that partitions applications between a processor and an intelligent memory subsystem. Active pages consist of a set of data and

associated functions that operate on that data. Essentially, the operations with this data is offloaded from the main processor to the active pages. In this sense, the concept of active pages bears similarity to the DSCP architecture proposed in this paper. However, our focus is on the current tightly-coupled multicore environments and the algorithms where the interactions between the data structure processing operations and the main algorithmic activities are frequent and very fine-grain in nature. In addition, implementing data structure offloading to a neigthboring core (or a thread context) requires only very minimal hardware support - much less than what is required for offloading computations to memory.

III. Data Structure Co-Processing

As described in the introduction, our objective is to increase the parallelism in sequential codes by off-loading data structure operations and executing them in parallel with the main algorithm, whenever possible. There are many data structures for which our approach is applicable. Binary search trees, red-black trees, heaps of various types all utilize basic operations of *insert*, *delete*, and a few types of queries, to maintain collections of data[4]. These operations typically take hundreds or even thousands of instructions (depending on the data structure size) to complete – large enough that it is worthwhile to look for ways to overlap processing, but small enough that low-overhead implementations are essential.

To evaluate the potential for offloading data structure processing, it's useful to guage how many cycles are typically consumed by data structure operations. We implemented a series of test programs which use C++ STL container classes to manage a variety of objects – and then profiled insert, delete, and search types of operations. We use the timing instruction of x86 ISA to gauge the number of cycles required for the execution of a given code.

We considered fourteen different STL container classes: stacks, vectors, maps, queues, double ended queues (deque), lists (implemented by the STL as doubly-linked lists), slists (singly linked lists), priority queues (implemented by the STL as a binary heap), multisets, multimaps, hash maps, hash multimaps, and hash sets. With each of these, we created collections that contained between one hundred and one hundred thousand elements, using integers, floats, strings, and complex objects with string and float fields.

The average number of processor cycles for element insertion, finding an extant object, finding a random (non-extant object), and removing an element, are shown in Tables I, II, III, and IV.

These results demonstrate that more complex data structures such as maps (which are typically implemented as red-black trees) require more work for each operation than a simple data structure such as a stack (although the number of cycles is non-negligible even for stack operations). Furthermore, these results show (not surprisingly) that the complex objects containing multiple elements require more work than simple integers or floating point numbers.

Data structures that induce an order on the elements result in much lower find times than data structures with randomly-ordered elements. As one might expect, insertion into a stack or a queue is relatively fast compared to the insertion into a hash map; removal or finding a particular element of a vector is a comparatively slow operation. There are instances where a particular operation is not appropriate for a container class – for example, one would not remove a randomly selected element from a priority queue; for this reason, we omit some columns.

Opportunities to extract parallelism depend heavily on the ratio of "processing work" and "data structure work." Ideally, these should be roughly balanced, with enough work in each transaction to avoid having synchronization dominate run times.

IV. DSCP Implementations

The basic approach of DSCP should be intuitive. We have investegated a variety of approaches to implement the idea.

A. ISA Support for DSCP

The maximum benefit of our approach is obtained when a processor instruction set is augmented with a few additional instructions. In simulations, we have added two new instructions: *DSCP_CONTROL rs, rt* and *DSCP_REQUEST rs, rt*. The *DSCP_CONTROL* instruction directs the main processor (through register rs) to either: activate a DSCP processor for a specific data structure (or to perform an operation), or deactivate it. The *DSCP_REQUEST* instruction stalls the calling thread from fetching further instructions until the desired request is satisfied.

When the DSCP receives an activation request (through *DSCP_CONTROL*), the request value is used to index into a table where the program counter for the appropriate code of the data structure can be found. This mechanism allows the DSCP to be updated with additional data structures or support additional changes (such as overloading the comparator), as may be required by the user. Once the request is generated, the memory for the data structure is allocated accordingly.

During the first instantiation of these routines on a multi-core implementation (data structure co-processor is a separate core), the DSCP will miss into its L1 cache and will have to fetch the instructions from the shared L2 cache, but since the instruction footprint of heap operations is very small, no further misses will be encountered after the cache is warmed up. A termination request through *DSCP_CONTROL* informs the DSCP when the heap is no longer needed so that its memory can be deallocated.

B. Compiler and Library Support for DSCP

To take advantage of additional instructions, the compiler requires slight modification. In our detailed experiments, we manually modified assembly language output of the compiler to include the additional instructions.

We envision the practical implementation of our approach through a modified data structure library (such as the C++

Size	set	stack	vector	map	queue	deque	list	slist	priority queue	multiset	multi map	hash map	hash m-map	hash set
Integers														
100	1816	116	136	1898	157	116	580	592	597	1484	1482	931	963	961
1k	1890	112	116	2072	110	106	593	583	421	1678	1669	921	952	969
10k	2511	106	121	2667	107	103	607	623	511	2237	2263	1030	1068	1089
100k	3094	114	139	3198	120	112	686	709	533	2803	2828	1046	1096	1115
Collection of doubles														
100	1720	136	116	1856	145	140	650	561	640	1495	1532	846	883	915
1k	1865	130	119	2122	127	126	606	516	479	1691	1724	830	877	914
10k	2543	133	120	2652	140	136	632	529	577	2253	2262	1060	1101	1110
100k	3092	133	124	3268	143	143	681	587	584	2807	2846	1071	1113	1126
Collection of strings														
100	2530	198	186	3136	201	189	680	623	799	2143	2473	1248	1330	1135
1k	2958	202	191	3573	196	194	673	621	1032	2592	2962	1480	1558	1331
10k	3963	212	228	4621	213	203	717	666	1398	3583	3992	1608	1661	1431
100k	5194	264	228	5976	236	227	749	697	1234	4838	5479	3024	2356	1917
Collection of complex objects														
100	4511	265	245	5897	263	256	759	683	3122	4402	7919	4086	1875	2179
1k	4818	270	255	4519	314	259	742	709	4106	4705	5222	5722	3831	3102
10k	7486	298	253	6340	287	268	1616	1573	5252	7394	7063	28953	25346	22765
100k	8472	310	282	9273	323	296	941	809	6297	8330	9821	415011	633278	450981

TABLE I

TYPICAL CYCLE COUNTS FOR INSERTION INTO A VARIETY OF C++ STL CONTAINER CLASSES.

Size	set	vector	map	deque	list	slist	multiset	multi map	hash map	hash m-map	hash set
Integers											
100	883	2764	1012	3914	4077	5650	912	911	561	554	541
1k	1013	16625	956	27772	27344	41690	1045	1005	423	423	398
10k	1469	129121	1391	267954	263843	423013	1479	1438	397	387	392
100k	1912	1280922	1852	2666224	2646558	4090299	1939	1890	462	439	447
Collection of doubles											
100	925	3132	905	3913	4612	5955	982	1005	544	541	517
1k	1063	21092	1029	26890	33424	44177	1081	1080	403	404	378
10k	1206	208733	1463	259116	324861	433203	1246	1240	504	493	464
100k	1993	2080462	1967	2584246	3253972	4341611	1965	2028	578	560	570
Collection of strings											
100	1204	9961	1246	10712	10652	12097	1234	2103	497	545	587
1k	1871	73829	1879	78523	80771	90135	1875	2394	616	625	635
10k	2692	712146	2726	769319	795923	889202	2693	3245	668	679	681
100k	3892	7122976	3959	7908804	8117235	8966676	3928	4813	1236	1423	1202
Collection of complex objects											
100	1895	12288	580	14277	14149	12106	1920	2417	746	699	772
1k	1981	92498	589	108845	104814	112230	2033	2425	2118	2437	2302
10k	2828	926674	2359	1053505	1025412	1096037	2767	3223	25112	23745	20560
100k	4660	9690556	4404	11836074	10907916	11761120	5078	4616	414917	601604	437183

TABLE II

TYPICAL CYCLE COUNTS FOR A FIND OPERATION A VARIETY OF C++ STL CONTAINER CLASSES. THE NUMBER OF ELEMENTS OF EACH DATA STRUCTURE IN EACH TEST RANGES FROM 100 TO 100 THOUSAND.

Size	set	vector	map	deque	list	slist	multiset	multi map	hash map	hash m-map	hash set
Integers											
100	847	2154	839	2448	2343	3613	884	1057	514	495	385
1k	1042	7377	996	11553	11174	15879	1059	1018	361	363	314
10k	1466	103728	1352	174704	172366	244720	1466	1403	493	498	487
100k	2041	563920	2318	952009	942763	1412416	2061	1974	492	477	456
Collection of doubles											
100	908	2508	883	2492	2717	4077	1013	943	437	357	335
1k	1072	9535	1046	11099	13628	17946	1101	1109	431	395	358
10k	1493	138168	1460	169151	211919	263735	1523	1487	404	394	380
100k	2117	749005	2162	922822	1157010	1455511	2122	2222	628	525	565
Collection of strings											
100	1400	5772	1391	5576	5192	6205	1422	1507	604	626	710
1k	2029	32078	2092	33714	34657	38678	2037	2198	683	699	657
10k	2859	505354	3004	533807	545070	624717	2841	3110	897	873	868
100k	4934	2949858	6396	3138526	3198838	3698628	4497	5730	1565	1718	1532
Collection of complex objects											
100	1646	8152	994	8862	8110	9092	2752	1686	1295	1211	1270
1k	2185	41190	1010	48317	45219	49884	2244	2148	1986	1914	1875
10k	3093	423835	2840	500765	485492	513286	3119	3158	16167	14877	13264
100k	5535	3030600	5178	3619087	3342308	3664937	5503	5510	265517	156397	146075

TABLE III

TYPICAL CYCLE COUNTS TO FIND A RANDOM ELEMENT IN A VARIETY OF C++ STL CONTAINER CLASSES.

Size	set	stack	vector	map	queue	deque	list	slist	priority queue	multiset	multi map	hash map	hash m-map	hash set
Integers														
100	3305	199	2913	2977	236	6718	1848	1138	1734	3824	3404	1539	1568	1457
1k	3249	154	8890	3446	169	29290	1676	1002	2017	4065	3497	864	838	794
10k	3797	136	110026	3668	157	318623	1946	1173	2435	4427	3695	972	986	967
100k	5509	147	657389	5272	148	1763622	2638	1522	3161	6794	7131	1215	1194	1165
Collection of doubles														
100	3318	216	3091	3579	196	7564	2184	1506	1593	3297	3612	1846	1902	1455
1k	3397	155	10709	4003	138	29945	2062	1398	2021	3735	4002	1231	1277	810
10k	3889	145	144761	4338	128	318286	2392	1669	2501	3908	4474	1545	1580	861
100k	5591	153	869950	7233	122	1758032	2997	2275	3254	6542	7701	2061	2077	1195
Collection of strings														
100	3903	435	8792	3706	494	11422	1541	1248	2829	4131	3769	1532	1505	1354
1k	4287	386	51473	4482	390	51170	1543	1199	3633	4300	4521	1347	1363	1154
10k	5361	400	591127	5576	402	743329	1889	1592	4872	5334	5705	1607	1583	1345
100k	9879	482	4762792	11927	482	4308748	4524	3692	6750	9345	11071	3055	3126	2586
Collection of complex objects														
100	2480	716	14835	2341	547	16287	1681	1400	6159	3692	2081	2440	2406	2540
1k	2840	599	107987	1782	467	100078	1697	1373	8145	2787	2345	4200	3868	3918
10k	3845	619	1046123	3064	493	973010	2077	1794	11380	3910	3289	26109	24044	21300
100k	8297	875	11296009	5603	637	7090246	6496	5752	14955	8503	5721	488533	496760	425564

TABLE IV

TYPICAL CYCLE COUNTS FOR REMOVAL OF AN OBJECT FROM A VARIETY OF C++ STL CONTAINER CLASSES.

STL), which would insulate the programmer from details. Ideally, the data structure library would recognize the hardware resources available, and take advantage of additional processing automatically.

C. Architectural Support for DSCP

The architectural support for DSCP depends on the specific architecture platform that is used. In this paper, we consider the DSCP implementation using two frameworks: (1) a Simultaneously Multithreaded (SMT) processor with the data structure co-processing implemented in a separate thread context (we assume a typical 2-way SMT processor for this study), and (2) a multicore processor with data structure co-processing implemented in a dedicated core. While other implementations are also possible, such as the use of a customized core for DSCP, these are beyond the scope of this paper.

1) SMT Implementation of DSCP: The first possible implementation of the DSCP paradigm is to use two thread contexts of an SMT processor to support the concurrent execution of the main algorithm, and the heap-related activities. The key advantage is that the first-level data cache is shared between the two threads, and therefore the cross-thread communication can be implemented directly using the cache storage, without requiring any additional hardware support. The major drawback of the SMT implementation is that the datapath resources (issue queue, register file, execution units) are shared by both threads, thus possibly impacting their performance (especially if rudimentary resource distribution policies are used) and imposing the limit on the performance potential that can be achieved through DSCP.

Although no additional hardware buffers are needed to pass the values between the two threads in the SMT implementation of DSCP, additional support is still required to properly synchronize the two activities. Specifically, the DSCP thread needs to know when a request is made. Similarly, the main thread needs to know when the value is returned from the data structure. One simple way of accomplishing that is to utilize simple software spin-locking on shared memory locations. This approach works without any changes to the ISA/compiler or the hardware. The spawn_heap call is automatically forking-off a DSCP thread in a free SMT context - the user is completely unaware of this action.

To avoid the overhead of spin-locking, either OS support can be used, or small hardware augmentations can be introduced. Specifically, a two-bit register can be added to the datapath to check the readiness of the information sent in both directions. The hardware can then check these bits every cycle to determine when the information coming from the heap is available for consumption, or when the new element is ready for heap insertion. We report the performance of this scheme in the next section.

2) Multicore Implementation of DSCP: Another possible implementation of the DSCP paradigm is through the use of two cores on a multicore processor. Instead of allowing the data structure helper thread to compete with the main process for CPU resources, a multicore processor permits the helper to run on a separate processor core which has its own set of resources. Multicores allow each thread to have the full complement of processor resources, therefore avoiding the resource contention. Unfortunately, the cores do not share the first level cache, and therefore shared-memory based communication is more expensive. It either requires an access to the L2 cache (write from the sender and read from the receiver), or relies on a cache coherence protocol to move the data among the L1 caches. In either case, this takes a longer time compared to similar communication through the shared caches on SMT.

V. EXPERIMENTAL RESULTS

To evaluate DSCP in a "real world" application, we on focus Dijkstra's shortest path algorithm, which utilizes a priority queue tightly integrated with distance update operations. We used used M-Sim 3.0 [12] - a significantly modified

version of the Simplescalar 3.0d simulator to obtain accurate measurements. M-Sim extends Simplescalar with models for both SMT and multi-core processors. Several versions of Dijkstra's algorithm were compiled on an Alpha 21264 using -O4 optimizations and then executed through the simulator. The simulated processor configuration is depicted in Table V.

Parameter	Configuration
Machine Width	4-wide fetch, issue and commit
Window Size	128-entry ROB, 48-entry LSQ
Functional Units and Lat (total/issue)	4 Int Add(1/1), 1 Int Mult(3/1) / Div (20/19), 2 Load/Store (1/1), 4 FP Add (2/1), 1 FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
Physical Registers	128 Integer + 128 FP Physical Registers
L1 I-Cache	32 KB, 2-way set-associative, 32 byte line, 1 cycle hit time
L1 D-Cache	32 KB, 4-way set-associative, 32 byte line, 1 cycle hit time
L2 Unified Cache	512 KB, 8-way set-associative, 128 byte line, 10 cycle hit time
Memory latency	300 cycles

TABLE V
CONFIGURATION OF THE SIMULATED PROCESSOR

As a first step, it's worthwhile to evaluate just how much performance improvement is possible with DSCP. To obtain this estimate, we separated the number of instructions (and CPU cycles) that were utilized by the heap-related operations, and those that were part of the main Dijkstra algorithm. For a typical sparse graph, averaging only a few edges per node, we find that heap operations dominate the work – roughly 60% of instructions (Figure 2) - and 50%-60% of processing cycles (Figure 3). These percentages are high, because an update for the distance to a vertex requires adjustment of the priority queue, which happens infrequently in these types of graphs.

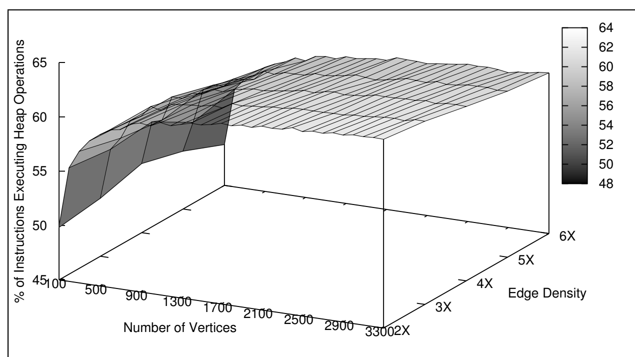

Fig. 2. Percentage of Instructions Executing Heap Operations in Sparse Graphs

In contrast, the work performed for dense graphs is dominated by the main Dijkstra algorithm. Each vertex has a large number of adjacent vertices, and the distances to each must be calculated. The priority queue must be updated only when the distance to a particular vertex improves – and this is relatively infrequent. Thus, while there is still gain possible, it is smaller than what is observed for the sparse graphs. This is illustrated in Figures 4 and 5 for instructions and cycles, dedicated to heap-related processing, respectively. As

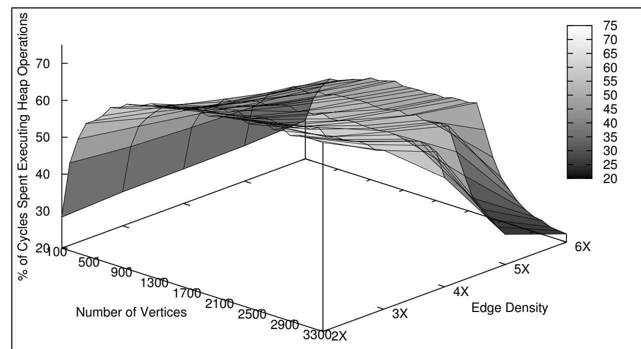

Fig. 3. Percentage of Cycles used for Heap Operations in Sparse Graphs

seen from these graphs, these percentages are much smaller, especially as the size of the graphs increases.

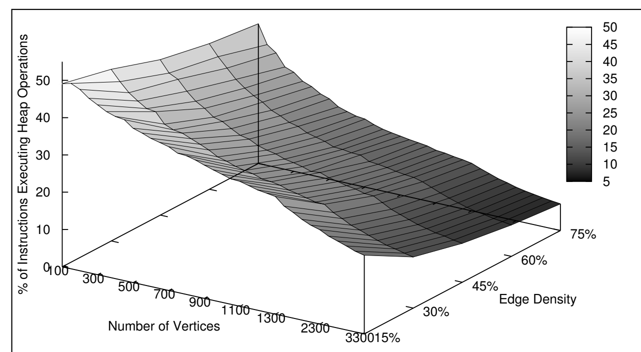

Fig. 4. Percentage of Instructions Executing Heap Operations in Dense Graphs

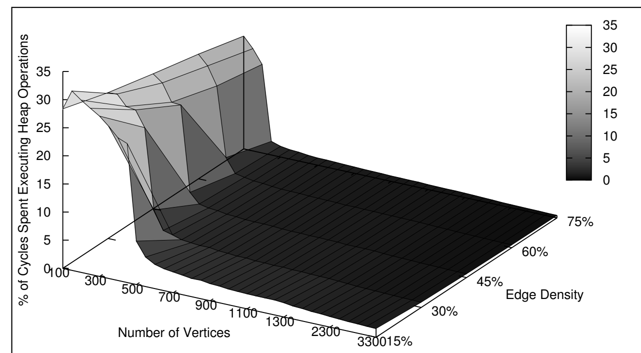

Fig. 5. Percentage of Cycles used for Heap Operations in Dense Graphs

While the ratio of heap instructions to non-heap instructions remains relatively constant in sparse graphs, the relative amount of time spent executing those instructions begins to drop with larger sparse graphs (Figure 3). In dense graphs, as shown in Figure 5, the time spent executing heap operations is small and therefore the expected benefit is also small.

These results are in some sense data structure and application specific. Other applications would obviously have a different mix of data structure work and processing within the main thread.

3) SMT Implementation of DSCP: Experiments for SMT processors utilized two threads, and spin-locking. Unfortu-

nately, the performance overhead of spin-locking (due to the extra instructions) is very large, especially taking into account resource sharing in SMT. Figure 6 depicts those results - for most of the graphs that we simulated, we actually observed performance losses, measured in the range of 5% to 15% compared to a simple single-threaded execution.

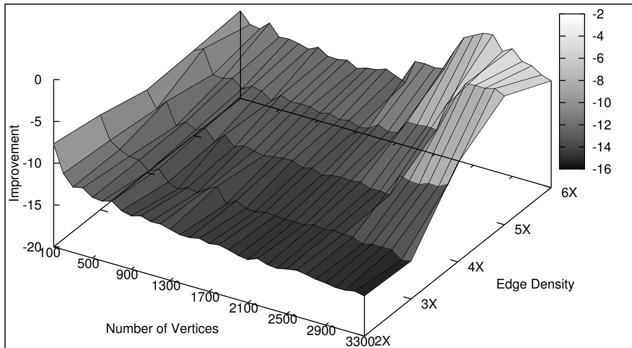

Fig. 6. DSCP Performance on SMT with Software Spin-Locking

4) Multicore Implementation of DSCP: For application to Dijkstra's algorithm, the multicore implementation of DSCP with hardware communication support was more successful. Figure 7 demonstrates the performance of this basic scheme. Due to the increased delays involved in passing values between the two threads (for these experiments we assumed a 10-cycle L2 cache latency and L2-based communication), significant performance losses are still observed. Performance losses are especially large for smaller data sets, and they are reduced for larger data sets, where the communication costs become less dominant. With cache-to-cache transfers supported as part of the coherence protocol, the performance will somewhat improve, but not all machines today provide this support.

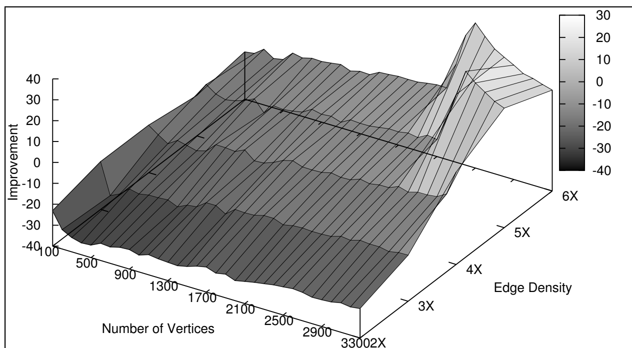

Fig. 7. Performance of Multicore-based Implementation of DSCP Without Hardware Support

The above results clearly demonstrate the need for architectural support to speed-up these communications. A very simple hardware support in the form of two registers, one supporting communication in each direction, is sufficient to address these latency challenges.

A. Improvements on Real World Graphs

In addition to experiments with synthetic graphs, we have also performed a series of tests using the DIMACS[5] benchmarks, which are road maps of the US and Europe. We observed improvements of up to 26% compared to a convential serial approach[15].

VI. Summary and Conclusion

As computing systems continue to rely more heavily on parallel computation as a means to boost performance, it is critical to find new opportunities. In this paper, we have focused on data structures, which are tightly integrated into a great many algorithms. Depending on the operation and data structure, there are opportunities to extract parallelism – the magnitude of gains possible depends on the specific problem addressed. With detailed experiments on Dijkstra's shortest path algorithm, we have obtained as much as 26% improvement over conventional serial processing – this problem is well known to be difficult to accelerate.

Separation of processing at the data structure level is attractive for many reasons. In particular, the changes are transparent to the programmer: by simply linking in a modified library, existing code may be accelerated.

Further acceleration is possible through the development of processor cores optimized for specific data structures; our experiments have revolved around utilizing ordinary SMT and multi-core platforms, but there is clear potential for additional gains with customized processors. The use of customized processors for DSCP, as well as implementations of other algorithms, are part of our current work.

References

[1] H. Akkary and M. Driscoll. A dynamic multithreading procesor. In *Proc. International Symposium on Microarchitecture (MICRO)*, 1998.
[2] G. M. Amdahl. Validity of the single-processor approach to achieveing large scale computing capabilities. In *Proc. AFIPS Conference*, pages 483–485, 1967.
[3] N. Clark, N. Hormati, and S. Mahlke. Veal: Virtualized execution accelerator for loops. In *Proc. International Symposium on Computer Architecture (ISCA)*, 2008.
[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithm, 2nd Edition*. MIT Press, 2001.
[5] 9th DIMACS implementation challenge: Shortest paths, 2006. Available online at: http://www.dis.uniroma1.it/∼challenge9/index.shtml.
[6] C. Madriles et al. Boosting single-thread performance in multi-core systems through fine-grain multi-threading. In *Proc. International Symposium on Computer Architecture (ISCA)*, 2009.
[7] R. Kumar et al. Heterogeneous chip multiprocessors. In *IEEE Computer, 38(11)*, 2005.
[8] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. International Symposium on Computer Architecture (ISCA)*, 1993.
[9] M. Hill and M. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, July 2008.
[10] E. Ipek, M. Kirman, N. Kirman, and J. Martinez. Core Fusion: Accomidating software diversity on chip multiprocessors. In *Proc. International Symposium on Computer Architecture (ISCA)*, 2007.
[11] C. Kim. Composable lightweight processors. In *Proc. MICRO*, 2007.
[12] M-sim: The multi-threaded simulator: Version 3.0, July 2009. Available online at: http://www.cs.binghamton.edu/∼msim.
[13] M. Oskin, F. Chong, and T. Sherwood. Active pages: A computation model for intelligent memory. In *Proceedings of the International Symposium on Computer Architecture*, 1998.
[14] R. Rajwar and J. Goodman. Transactional lock-free execution of lock-based programs. In *Proc. 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.
[15] Name Removed. Title removed. *Removed For Blind Review*.

[16] J. Renau, K. Strauss, and L. Ceze. Energy-efficient thread-level speculation on a CMP. *IEEE Micro*, 26(1), January/February 2006.

[17] A. Roth and G. Sohi. Speculative data-driven multithreading. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2001.

[18] P. Salvedra and C. Zilles. Fundamental performance challenges in horizontal fusion of in-order cores. In *Proc. HPCA*, 2008.

[19] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. *Proc. International Symposium on Computer Architecture (ISCA)*, 1995.

[20] M. Suleman, O. Mutlu, M. Qureshi, and Y. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[21] H. Zhong, S. Lieberman, and S. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proc. International Symposium on High Performance Computer Architecture (ISCA)*, 2007.

[22] C. Zilles and G. Sohi. Understanding the backward slices of performance degrading instructions. In *Proc. International Symposium on Computer Architecture (ISCA)*, 2000.

[23] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proc. International Symposium on Computer Architecture (ISCA)*, 2001.