Customized Architectures for Faster Route Finding in GPS-Based Navigation Systems

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

GPS based navigation systems became popular in dedicated handheld devices, and are now also found in modern cell phones, and other small personal devices. A key element of any navigation system is fast and effective route finding, and this depends heavily on Dijkstra's shortest path algorithm.

Dijkstra's algorithm is serial in nature; prior efforts to accelerate it through parallel processing have had almost no success. In this paper, we present a practical approach to extract small-scale parallelism by shifting priority queue operations to a secondary tightly-coupled processor. We obtain a substantial speedup on real-world graphs (in particular, road maps), allowing the development of navigation systems that are more responsive, and also lower in total power consumption.

I. Introduction

The availability of global positioning system (GPS) receivers has resulted in the growth of navigation as a key element of many portable devices. Many people now rely on automatic route finding applications in cell phones, or the ones built into automobile electronics.

A key element of route finding is Dijkstra's shortest path algorithm[5]. This is an inherently serial algorithm, which has been proven difficult to accelerate. In this paper, we present an approach to speed up the algorithm by offloading priority queue management to a secondary processor and performing the data structure bookkeeping operations in parallel with main algorithmic activities.

This work is part of a larger effort to alleviate sequential bottlenecks. 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 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 the well-known, but often forgotten, Amdahl's Law [1]. 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 a multicore chip [11].

For Dijkstra's algorithm, the interface between the main loop and the priority queue provides a "clean break" between two activities, allowing overlapping of work with minimal bookkeeping overhead. Obtaining parallelism by offloading portions of work is by no means a new concept in general. Our focus is on the interface between fine-grain data structure operations and algorithmic work - to the best of our knowledge offloading the work at this level is a novel approach. We demonstrate that modest hardware support can provide a significant performance advantage over a software-only approach implemented on commodity hardware. We explicitly take advantage of the limited range of possible operations on a data structure (essentially insertion, deletion, and queries) to enable a high performance, low overhead interface.

We refer to the overall strategy of offloading data structure operations as "data structure coprocessing" (DSCP). We focus on an implementation for a priority queue in this paper.

We evaluate the proposed idea using the DIMACS shortest path implementation challenge benchmarks [6] (as well as a range of synthetically-generated graphs); it should be noted that all of the competitive approaches for these benchmarks are essentially serial. The lack of any meaningful parallel speedup (much less, anything approximating scalable speedup) highlights this problem as a hard serial bottleneck, and an important area to achieve improvements for application-level speedups. The DIMACS challenge benchmarks consist of road maps for the United States and Europe; our DSCP approach can achieve speedup well beyond what traditional methods obtain, and with a very modest cost in terms of hardware.

We demonstrate that on average across all simulated synthetic sparse graphs with different number of vertices and edge densities, a multicore implementation of DSCP on Dijkstra's algorithm achieves a 26% performance improvement compared to traditional serial execution. For sparse graphs, there still remains significant room for future improvements and we suggest avenues for closing this gap. We further demonstrate that the range of performance improvements for the complete USA road map is between 20% to 25% depending on the size of the L2 cache used.

II. Single Source Shortest Paths

As our experiments in this paper revolve around the computation of shortest paths using Dijkstra's algorithm [5], we now describe this algorithm briefly and discuss related work.

A. Dijkstra's Algorithm

Algorithm 1 illustrates the well known approach for computing shortest paths. This is based on the implementation by Cormen[4], with edge relaxation integrated in.

Implementations of Dijkstra's algorithm repeatedly perform edge relaxation operations. The distance to any vertex i is known as $d[i]$; if there is an edge between vertices u and v with length $w(u,v)$, then $d[v] \leq d[u] + w(u,v)$. Dijkstra's algorithm uses the priority queue to consider each vertex, ordered by distance from the starting vertex. Each time an improved distance is found to some vertex v, the position of v in the priority queue must be updated.

Algorithm 1 Pseudocode for Dijkstra's shortest path algorithm.

```
Initialize-Single-Source(G,s)
S ⇐ 0
Q ⇐ V[G]
while Q ≠ 0 do
    u ⇐ Extract-Min(Q)
    S ⇐ S ∪ u
    for vertex V ∈ Adj[u] do
        if d[v] > d[u] + w(u,v) then
            d[v] = d[u] + w(u,v)
            Update-Queue(Q,v)
        end if
    end for
end while
```

For Dijkstra's algorithm, there are many efficient methods to implement a priority queue (for example, [22], [2]). The specific data structure and implementation, however, is orthogonal to the focus of our work. We are interested in demonstrating a practical approach to accelerating Dijkstra's algorithm with customized architectures. Our experiments use a simple binary heap to illustrate our approach.

Insertion of a value into a heap is illustrated in Algorithm 2. The variable *heapsize* tracks the number of elements in the heap, while $LEFT(i)$, $RIGHT(i)$, and $PARENT(i)$ are simple macros to refer to the left child, right child, and parent of node i. The array H is used to contain the heap.

Algorithm 2 Insertion into a priority queue based on a simple binary heap.

```
H[heapsize] = new_value
i = heapsize
heapsize = heapsize + 1
while i > 0 do
    if H(PARENT(i)) > H(i) then
        swap H[PARENT(i)], H[i]
        i = PARENT(i)
    end if
end while
```

In our implementation, the heap contains both the distance value (which is used to order the heap), and the associated vertex number. The amount of work done to insert a new value into a heap is at worst logarithmic with the number of elements. For large graphs, there can be a significant number of iterations of the while loop. One may improve the above implementation by also supporting *DECREASE_KEY* operations; there are a wide range of variations possible, each with their advantages and disadvantages.

The EXTRACT-MIN operation performed in Dijkstra's algorithm operates by removing the top element, and then moving a leaf element into the top location. Heap order is restored by trickling down the top element, illustrated in Algorithm 3.

Algorithm 3 Extraction of a minimum value from a priority queue based on a simple binary heap.

```
if heapsize = 0 then
    return empty
end if
return_value = H[0] {Main thread may continue.}
heapsize = heapsize - 1
H[0] = H[heapsize]
i = 0
while i < heapsize do
    largest = i
    if LEFT(i) < heapsize and H[LEFT(i)] > H[largest] then
        largest = LEFT(i)
    end if
    if RIGHT(i) < heapsize and H[RIGHT(i)] > H[largest] then
        largest = RIGTH(i)
    end if
    if largest ≠ i then
        swap H[i], H[largest]
        i = largest
    else
        i = heapsize
    end if
end while
```

We note that the return value for an EXTRACT-MIN operation is available immediately. We assume that this value is returned to the main algorithm as quickly as possible, with the update to the priority queue data structure overlapping with the other work. EXTRACT-MIN operations also require work that is logarithmic with the size of the problem.

B. Parallel Approaches to Shortest Paths

As noted in the introduction, there has been a major shift from more complex serial processing to parallel computatation with large numbers of processor cores. We wish to highlight that the shortest path problem is in fact a hard serial bottleneck; while this problem has been studied in detail, no efficient and effective parallel approach has been proposed until now.

Dijkstra's algorithm is not the only approach to finding a shortest path. The Bellman-Ford algorithm is also well known, and remarkably simple to implement. The edge relaxation operation used by Dijkstra can be applied en masse, making a parallel (or vector based) approach easily applicable.

In [9], for example, a general purpose graphics co-processor (GPGPU) was used to implement the Bellman-Ford algorithm. While tremendous gains compared to a serial Bellman-Ford implementation were achieved, the method is competitive with Dijkstra only on small graphs. The computational complexity of the Bellman-Ford algorithm is $O(V * E)$, where there are V vertices and E edges. Compared to the well known Dijkstra algorithm, which has a complexity of $O(E + V\log V)$, the Bellman-Ford approach is inefficient.

Another attempt to speed up shortest path computations was made by a team of researchers using a Cray supercomputer [15]. In some respects, the approach is a hybrid of the Dijkstra and Bellman-Ford algorithms. In the Dijkstra approach, the priority queue forms a serial bottleneck, restricting computation to consider only a single u vertex at any given time.

Their Δ-stepping algorithm performed speculative distance calculations, with alternating phases of computation and correction. Vertices are ordered into "buckets," and are processed in parallel. If the distance to any vertex in a bucket remains unchanged, then this speculative computation is effective, and the problem can be solved more swiftly. If the distance to a vertex is updated, the vertex must be reprocessed, and the additional work is wasted.

For a set of carefully constructed graphs, the approach provided performance gains – in realistic situations, however, one cannot predict graph structure. On experiments with graphs such as road maps, the approach was much slower than a conventional sequential approach. The authors noted that obtaining scalable performance improvements for the shortest path problems remains an open challenge.

C. Architectures for Shortest Paths

To date, Dijkstra's shortest path algorithm remains the most efficient implementation. In order to accelerate this algorithmic approach, we propose to separate the work performed with data structures from the work of the main algorithm. In many instances, work can be performed simultaneously and asynchronously, resulting in speedup with minor hardware costs and few changes to the original application code. This is illustrated for a small graph in Figure 1. To be specific, the following occurs:

- The main processor performs an Extract-Min operation; the top element of the heap is returned *immediately* by the DSCP.
- While the main processor begins to update distances to other vertices, the DSCP restores order to the priority queue.
- Computations of distances and updates to the priority queue are decoupled, allowing simultaneous processing.

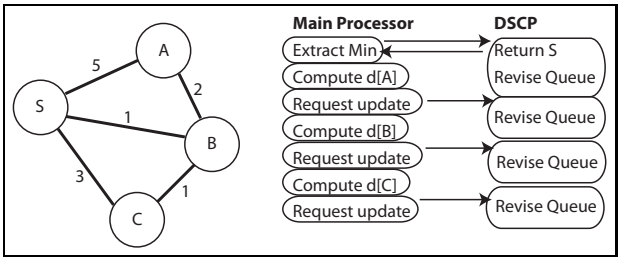


Fig. 1. Shortest path in a graph, and the opportunity for accelerating Dijkstra's algorithm with a data structure co-processor.

III. Evaluation Methodology

In order to determine the performance impact of the schemes described in this paper, we used M-Sim 3.0 [14] - a significantly modified version of the Simplescalar 3.0d simulator. M-Sim extends Simplescalar with models for both SMT and multi-core processors. Several versions of Dijkstra's algorithm (Section II-A) were compiled on an Alpha 21264 using -O4 optimizations and then executed through the simulator (with some optimizations applied to the assembly code, as detailed in Section 4.3). The detailed simulation results were collected from the point where the program sets the parameters for the first heap insertion (after all data has been stored in memory) to the point of the program's normal termination.

To compare performance, we used the actual number of cycles required to complete both algorithms on the simulator. The simulated processor configuration is depicted in Table III.

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 I
CONFIGURATION OF THE SIMULATED PROCESSOR

A number of studies were performed with synthetic graphs to determine the scope of possible gains. In this paper, we present results of road map benchmarks from the 9th DIMACS Implementation Challenge [6]. These are standard benchmarks for shortest path algorithms, and contains distances for major roads and cities in the USA. Experiments with European road maps produce similar results.

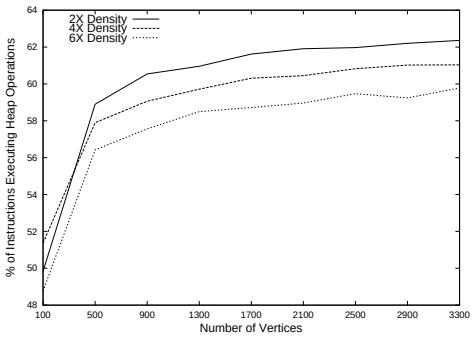
We compare our performance improvements against the baseline implementation which executes Dijkstra's algorithm as described in Section II-A on a single threaded superscalar processor configured according to Table III.

IV. Faster Shortest Paths

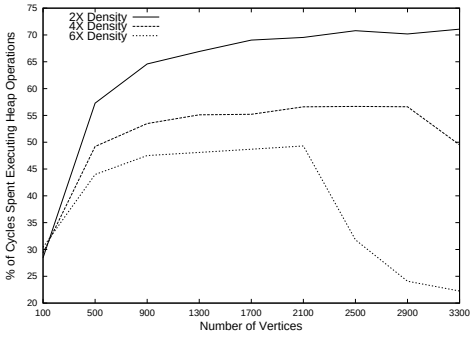
The merits of offloading priority queue operations depends on how much work is performed by the data structure, compared to the main algorithmic thread. Using synthetic sparse graphs, we find that this work is roughly balanced; for a variety of small graphs, with between 2X and 6X the number of edges and vertices, we find the number of instructions, and the number of compute cycles required, to be roughly comparable. This is illustrated in Figure 2.

We would note that many practical graphs (road maps, most computer networks, and so on) are sparse. Any graph which has primarily local connections will be sparse. For example, the DIMACS Full USA road map graph that we evaluate in this paper has an average of 2.43 edges for each vertex. The separation of work into heap and non-heap effort provides a quick upper bound on performance gains possible if the work is divided among two processors. Ideally, the work could be evenly divided, and for sparse graphs, we are not far from this.

In the rest of this section, we describe the modest changes needed to implement the overlapping of heap management operations with the main algorithm. Specifically, we demonstrate a DSCP-aware high-level language implementation of Dijkstra's algorithm (using *C*), describe the compiler and



(a) Instructions



(b) Cycles

Fig. 2. Percentage of a) Instructions or b) Cycles used Executing Heap Operations in Sparse Graphs

ISA support, and then describe hardware support needed to efficiently execute the modified code on a multicore processor.

A. DSCP-Aware High-Level Language Implementation

Traditional implementations of Dijkstra's algorithm, exemplified by Section II-A, are not easily amenable to parallelization, because the main algorithmic functions (basically, the distance vector update based on the values returned from the heap) are tightly integrated with the heap management operations. The first step in providing a clean separation of the two activities is to decouple their address spaces and provide routines that trigger explicit bi-directional communication between the main algorithm and the data-structure related operations.

A number of new routines are provided. Specifically:

- spawn_heap(int num_elements): Initialize the DSCP using num_elements to determine the heap size. Communication space is created and the co-processor begins fetching heap related code. If the co-processor is not available, the code terminates with an error. Otherwise, it unblocks the main process. The DSCP allocates memory for the heap and waits for requests.
- heap_insert_request(int vertex, int distance): Waits for the communication buffer to be empty (as explained later), then sends the data to be inserted (vertex, distance) to the co-processor. Implementation details of this communication are described in the following subsections.
- heap_pop_request(int *storage): Waits for the communication buffer (again, explained later) to be empty. It then checks if the heap is empty, and if it is, 0 is immediately returned before actually performing the heap pop operation. Otherwise, heap pop is performed and the returned value is copied into the memory location indicated as an argument.
- unspawn_heap(): Uninitializes the heap.

These changes to the Dijkstra's algorithm's code are minimal and do not require the programmer to have knowledge of the underlying DSCP implementation. Library support can easily make this code portable and allow it to utilize whatever underlying support is available - if any.

B. ISA Support for DSCP

DSCP requires two new instructions to be added to the ISA: *DSCP_CONTROL rs, rt* and *DSCP_REQUEST rs, rt*. The main processor is directed by the *DSCP_CONTROL* instruction (through register rs) to either: a) Activate the DSCP as a heap co-processor with a heap size specified by register rt or b) Deactivate the DSCP.

The *DSCP_REQUEST* instruction may block the calling thread from fetching further instructions until the desired request is satisfied. Some of the requests supported by these instructions are shown in Table IV-B and explained in more detail in the next subsection.

Request	Value of rs	Blocking	Value of rt
Insert Vertex	1	No	Data to DSCP
Insert Distance	2	No	Data to DSCP
Heap Insertion	4	No	NULL
Heap Pop	8	Yes	Data from DSCP
Early Pop Resolution	16	Yes	Data from DSCP
Pop Answer	32	Yes	Data from DSCP
Service Request	64	Yes	Data from DSCP
Make Request	128	Yes	NULL

TABLE II
DSCP_REQUEST PARAMETERS

During the first instantiation of these routines on a multicore implementation, 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. Synchronization details between the main algorithm and the DSCP depend on the actual implementation, and are described in the later sections.

C. Compiler and Library Support for DSCP

In the next step, we compile the code modified with the routines specified in Section IV-A on the Alpha AXP ISA using the native DEC C compiler. The assembly language output of the compiler was then modified by hand (a compiler can be easily modified to perform these modifications automatically) to insert the new instructions specified in section IV-B and also perform a number of optimizations. These changes largely

involved placing the new instructions described in section IV-B and also manually generating the code to decide which heap functions to execute based on the incoming requests. Normally, the latter would be provided in the form of library support. Specifically, the following translations from the new high-level routines described in Section IV-A to the new ISA instructions were performed:

```
spawn_heap(int num_elements):
    DSCP_CONTROL Create_Heap, num_elements;
    return 1;
heap_insert_request(int vertex, int distance):
    DSCP_REQUEST Make_Request, NULL;
    DSCP_REQUEST Insert_Vertex, vertex;
    DSCP_REQUEST Insert_Distance, distance;
    DSCP_REQUEST Heap_Insertion, NULL;
heap_pop_request(int *storage):
    int early_pop;
    DSCP_REQUEST Make_Request, NULL;
    DSCP_REQUEST Early_Pop_Resolution, early_pop;
    if(early_pop == 1)
        return 0;
    DSCP_REQUEST Heap_Pop, *storage;
    return 1;
unspawn_heap():
    DSCP_CONTROL Stop_Heap, NULL;
    return 0;
```

The actual heap code and supervisory code are maintained on the DSCP - these code segments can be enhanced to provide additional functionality by supporting other data structures. The interface provided by the *DSCP_CONTROL* instruction easily allows for such extensions.

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.

D. Multicore Implementation of DSCP

While it is possible to implement DSCP using either a separate thread of an SMT processor or a seperate core of a multicore processor, in this paper we only present the multicore implementation. In this case, instead of allowing the data structure helper thread to compete with the main process for CPU resources (as would be the case with SMT), 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. On the flip side, the cores do not share the first level cache, and therefore shared-memory based communication is more expensive9agaon, compared to possible SMT-based implementation). 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. Due to significant delays involved in passing values between the two threads (we assumed a 10-cycle L2 cache latency and L2-based communication), considerable performance losses are 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.

Communication delays clearly demonstrate the need for architectural support to speed-up interprocess communication. Very simple hardware support in the form of two registers, one supporting communication in each direction, is sufficient to address these latency challenges.

To_DSCP register is used for sending information from the main processor to the DSCP. *From_DSCP* register is used to send the information from the DSCP back to the main processor.

The *To_DSCP* register is composed of the following fields:

- *Request Type*: The request type field requires 2 bits. The first bit indicates the type of request (Insert or Pop), and the second bit is used to clear the register.
- *Insertion Data*: The vertex and associated distance value for heap insertion. The size of this field is two memory words (128 bits).
- *Clear Pop Done*: This flag is cleared by the main processor after it uses the *Pop Answer* value returned by the DSCP.
- *Controls*: This is used to enable, disable, or insert code into the DSCP.

The *From_DSCP* register is composed of the following fields:

- *Clear Request*: Clearing this field allows another request to be placed.
- *Early Pop*: Contains the return value of Heap_Pop if the heap is empty.
- *Pop Answer*: Returns the value produced by Heap_Pop. This is a 64-bit value.
- *Pop Done*: Informs the main thread that Heap_Pop operation is completed and the result is ready.

These communication registers can be used to provide support for a wide range of data structures. In this paper we constrain our discussions to regular binary heaps.

Using the communication mechanism based on two inter-core registers as described above, significant performance improvements can be realized, as we demonstrate in Section V. However, since only one register is present in each direction, the main thread must wait for its request to be acted upon by the DSCP before it can make another request, thus limiting parallelism to a single request. This potentially performance-limiting situation can be mitigated by using a request queue in an effort to accept incoming requests and clear the *Request_Type* field - allowing more requests to be issued to the DSCP (but not serviced) in parallel. This solution would only apply for heap insertions, but can not be used for heap pop operations, because when a heap pop request is made, the main thread cannot continue without receiving the result.

Furthermore, the implementation of a heap provides an opportunity for another optimization that allows faster servicing of heap pop requests. Specifically, the heap insertions are not order dependent. Elements issued for insertion between two successive pops can be actually inserted into the heap in any order. However, heap pop operations can not be serviced until all pending heap insertions have been processed. By allowing the request queue to compare the pending heap insertion values and the top of the heap, it is possible to determine if a pending heap insertion would become the smallest value in the heap or if the heap already has the smallest value (assuming min-heaps). If the heap has the smallest value, the heap pop can be moved to the front of the queue so it can pick up the result without waiting for heap insertions that will not change the heap. If a pending insertion contains the smallest value, the heap pop can be serviced immediately by providing the value from heap insert directly to heap pop - this would eliminate of both a heap pop and heap insert.

We implemented both of these optimizations, but found the additional benefits to be very small. More details on this is provided in Section V-B.

V. Results and Discussions: Synthetically Generated Graphs

In this section, we present the results of our experiments with various DSCP implementations and analyze the performance results.

A. Potential Performance Benefits and Impact of Perfect Caches

The goal of our first set of experiments was to estimate an upper bound on possible performance improvements with DSCP. We estimated this bound using the following methodology. We replaced each function call for accessing the heap with a no-OP instruction and placed the correct results (obtained from a previously constructed trace file) in the appropriate registers or memory locations. This provides the illusion of having instantaneous access to the heap and therefore estimates the upper bound on the performance gains achievable by deploying the DSCP framework.

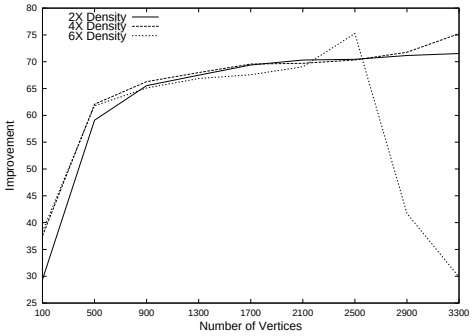


Fig. 3. Upper Bound on Performance Improvement with DSCP

The results of the experiment for sparse graphs are presented in Figure 3. Improvements are above 30% in all cases, generally rising as the number of vertices increases and leveling off around 70%. With the largest graphs, theoretical performance benefits drop with the largest edge density (6X): for 2600 nodes - 63.04%, for 2700 nodes - 54%, for 2800 nodes - 47%, for 2900 nodes - 42%, for 3000 nodes - 38%, and for 3300 nodes - down to 30%. A reduction in potential improvements can also be observed for large graphs at 5X edge density.

For graphs with a high ratio of edges to vertices (approaching dense graphs), the main algorithmic loop of Dijkstra's algorithm performs a great deal of work without needing to update the heap – and thus, the potential benefit of parallel computation is reduced. For most sparse graphs, however, the work is reasonably balanced.

B. Evaluating DSCP with Architectural Enhancements

Next, we present the experiments evaluating the performance of DSCP using a multicore implementation with inter-core communication registers. Specialization of the processor architecture can provide significant benefit.

Figure 4 shows the difference between the maximum potentially realizable performance gains (assuming instantaneous accesses to the data structure) and the actual gains achieved by DSCP implementation on multicore.

On the average across all simulated graphs, with multicore implementation additional 40% gains are still on the table. These additional gains can be realized by overlapping multiple insert operations (that is, supporting simultaneous insertions of several elements into the heap), but that requires the design of a specialized core specifically oriented towards such task. Investigations of such a design are beyind the scope of this paper and are left for future work. In this paper, we focus on schemes that require minimal hardware changes.

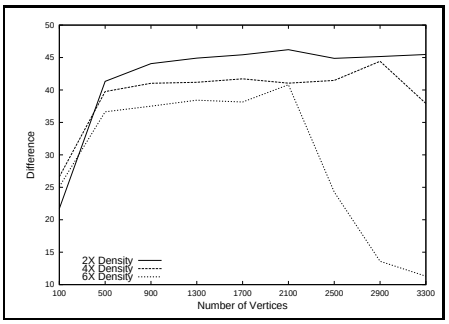


Fig. 4. Performance Improvement for Multicore and SMT Implementations of DSCP Versus Optimal

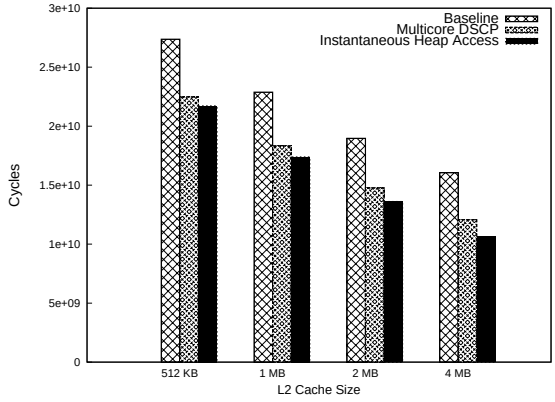
Finally, we evaluated the use of hardware queues between the cores to allow issuing of multiple requests from the main processor to DSCP. We found that having a single communication register in each direction is sufficient for extracting most of the performance potential. This is because the amount of time between successive heap pop operations is sufficiently large to amortize the amount of time spent waiting for injecting the insertions into the queue. We also evaluated the request reordering mechanism described in Section IV-D and found that it adds only another 0.1% performance improvement on the average, which is negligible. Based on these experiments, we conclude that these advanced mechanisms are not needed and a simple communication interface through a pair of registers is sufficient.

VI. Results and Discussions: Full USA Benchmark

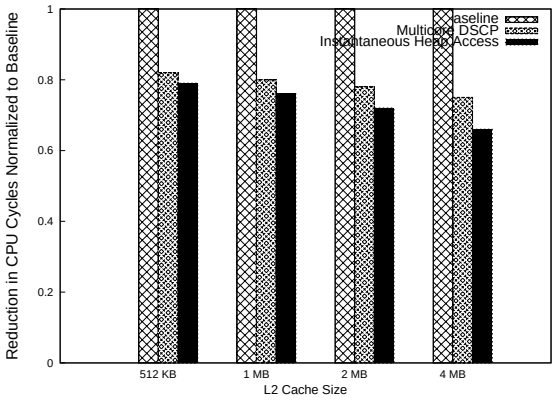
In this section, we evaluate the performance of DSCP on a large real-world benchmark - a Full USA roadmap, provided by the Center for Discrete Mathematics & Theoretical Computer Science [6]. This graph contains nearly 24 million vertices and just over 58 million edges. Just to illustrate the scale of this benchmark, discovering the shortest path took almost 2 minutes to complete on a 2.33 GHz Intel Xeon machine.

As seen in Figure 5(a), the maximum potentially realizable performance benefit for the Full USA map ranges from 20.7% (for 512KB L2 Cache) to 33.9% (for 4MB L2 Cache). Clearly, as the capacity of the L2 cache increases and the memory-related delays consequently lessen, DSCP performance gains

increase, as heap-related processing now presents a relatively larger bottleneck.



(a) In CPU Cycles



(b) Normalized to the Baseline Performance

Fig. 5. Execution Time of the Full USA Benchmark for Different L2 Cache Sizes.

Multicore implementations of DSCP result in an 18% performance improvement for 512KB caches and upwards of 25% with 4MB L2 caches. Notice that with larger L2 cache sizes the window of opportunity for the DSCP scheme widens. There is more potential to exploit additional mechanisms for speeding up remaining data structure related bottlenecks. These results are consistent with the results for synthetic graphs presented in the previous section.

Smaller real-world benchmarks are shown in Figure 6 (6(a) - New York, 6(b) - Northeast USA, 6(c) - Florida). The same general trends are realized with these benchmarks as compared to the Full USA benchmark and our synthetic benchmarks. Realizable performance ranges from 29-31% with 1MB L2 caches, 31-32% with 2MB L2 caches and 38-40% with 4MB L2 caches. Multicore implementations of DSCP obtain approximately the same performance benefit as the Full USA benchmark (with the Florida benchmark performing about 1% better).

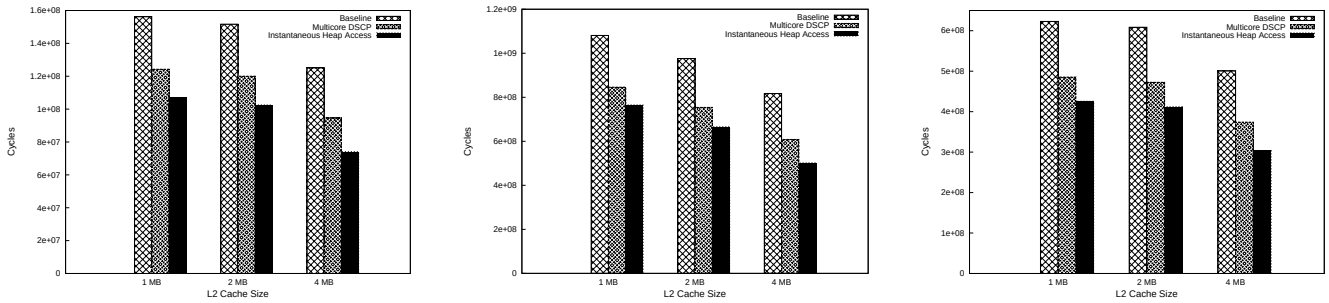
VII. Related Work

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 [8], [11], [21], core fusion techniques [12], [13], [19], speculative multithreading and pre-execution [7], [17], [20], [24], [25], [18], 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 [12] 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 [19] 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 [13]. The TRIPS architecture requires special ISA and the compiler support. Simple aggregation of multiple cores only provides a larger and more powerful processing core, but does not address 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. Our proposal addresses this fundamental bottleneck.

Asymmetric (or heterogeneous) multicore architectures represent one approach for efficiently executing performance-critical portions of the program [8], [11], [21]. In the approach of [21], 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 [10], [16]. The critical sections used in parallel applications (which are the target of the work of [21]) 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 [23], 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.



(a) New York (b) Northeast USA (c) Florida

Fig. 6. Execution Time of various DIMACS Benchmarks for Different L2 Cache Sizes

VIII. Concluding Remarks and Future Work

The importance of navigation should not be underestimated. Traffic information is now available in real-time, and there is a great reliance for fast, accurate guidance. The single most time intensive component is Dijkstra's algorithm, and to our knowledge, this work is the first to obtain a signficant performance gain on real-world maps through the use of additional processing resources. The major contributions and the key results of this paper are the following:

- We described the language, compiler, ISA and architecture support for DSCP.
- We demonstrated the application of the DSCP paradigm to Dijkstra's shortest-path algorithm and showed the realization using multicore architectures.
- We evaluated the performance benefits of DSCP using both synthetically generated dense and sparse graphs, and the real-world application (full road map of the United States).
- We demonstrated that significant performance improvements (up to 25% for the USA map) can be realized with DSCP with minimal hardware support, compiler and ISA modifications.

Our future work will examine the use of customized hardware accelerators for data structure co-processing. One of the ideas is to provide hardware support for overlapping multiple heap insert operations. There is nothing that restricts our approach to either Dijkstra's algorithm or binary heaps; we expanding our work to include other algorithms and data structures. We are also collaborating with industry colleagues to evaluate the DSCP paradigm within synthesizable microprocessor designs.

References

[1] G. M. Amdahl. Validity of the single-processor approach to achieveing large scale computing capabilities. In *Proc. AFIPS Conference*, pages 483–485, 1967.

[2] B. V. Cherkassky, A. V. Goldberg, and C. Silverstein. Buckets, heaps, lists, and monotone priority queues. In *Proc. Symposium on Discrete Algorithms*, pages 83–92, 1997.

[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] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[6] 9th DIMACS implementation challenge: Shortest paths, 2006. Available online at: http://www.dis.uniroma1.it/∼challenge9/index.shtml.

[7] 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.

[8] R. Kumar et al. Heterogeneous chip multiprocessors. In *IEEE Computer, 38(11)*, 2005.

[9] M. Garland. Sparse matrix computations on manycore GPU's. In *Proc. Design Automation Conf*, pages 2–6, 2008.

[10] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. International Symposium on Computer Architecture (ISCA)*, 1993.

[11] M. Hill and M. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, July 2008.

[12] 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.

[13] C. Kim. Composable lightweight processors. In *Proc. MICRO*, 2007.

[14] M-sim: The multi-threaded simulator: Version 3.0, July 2009. Available online at: http://www.cs.binghamton.edu/∼msim.

[15] K. Madduri, D. Bader, J Berry, and J. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[16] 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.

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

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

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

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

[21] 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.

[22] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, May 1999.

[23] 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.

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

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