Operating System Support for Shared Hardware Data Structures

by Gedare Bloom

B.S. in Computer Science and Mathematics, May 2005, Michigan Technological University
M.S. in Computer Science, August 2012, The George Washington University

A Dissertation submitted to

the Faculty of
School of Engineering and Applied Science
of The George Washington University
in partial satisfaction of the requirements
for the degree of Doctor of Philosophy

January 31, 2013

Dissertation directed by

Bhagirath Narahari
Professor of Engineering and Applied Science and of Engineering Management & Systems Engineering
and
Rahul Simha
Professor of Engineering and Applied Science

The School of Engineering and Applied Science of The George Washington University certifies that Gedare Bloom has passed the Final Examination for the degree of Doctor of Philosophy as of November 19, 2012. This is the final and approved form of the dissertation.

Operating System Support for Shared Hardware Data Structures

Gedare Bloom

Dissertation Research Committee:

Bhagirath Narahari, Professor of Engineering and Applied Science and of Engineering Management & Systems Engineering, Dissertation Co-Director

Rahul Simha, Professor of Engineering and Applied Science, Dissertation Co-Director

Gabriel Parmer, Assistant Professor of Computer Science, Committee Member

Evan Drumwright, Assistant Professor of Computer Science, Committee Member

Guru Prasadh Venkataramani, Assistant Professor of Engineering and Applied Science, Committee Member

Dedication

For my grandfather Laird, who inspired me to seek higher education.

For my wife Veronica, who inspires me to better myself.

For my daughter Annalise, who inspires me to better the world.

Acknowledgments

Thank you Veronica, for challenging me to improve in all aspects of life. I love you.

No one is an island. I am grateful for all the assistance I have received throughout my life. A dissertation is the culmination of a long journey, an epic quest of self-discovery that starts when the young mind is planted with the seed of introspection. Along the way, many hands help to sow the seed and till its soil, and to the people whose hands have helped me, I am grateful.

To my parents, Uno and Jodi, for encouraging academic success, praising hard work and good grades, permitting my obsessive reading, and for chasing their dreams. To my siblings, Jeni and Adam for enduring and passing lessons learned, and to Ric and Ben for following and providing me with retrospective. To my grandparents, Laird and Marcia Heal, Elsa Bloom, and Beulah Huff, whose memories I treasure, for instilling in me the virtue of being studious, wholesome, and hard-working: *mens sana in corpore sano*.

To my aunts and uncles: to Dicky for embodying sisu; to Sandy Martin for introducing me to science, to Kathy Soderbloom for a larger world of politics and religion, to Diana Anderson for intellectual challenges and inspirations; to David, Andy, and Loren Heal for introducing the digital world to me, to Bud Heal for maintaining some of the old world, and to Kim Rosser, who always seems positive to me.

To all the wonderful teachers and professors who are *there* for their students.

To Mrs. Weber, my 5th grade science teacher who first introduced me to controlled scientific experimentation.

To Mr. Wang, my 7th and 8th grade math teacher, for seeing in me a skill for math and encouraging me to develop it beyond the course material, and for the occasional pick-up basketball game, in which I got to socialize with a teacher outside the confines of the classroom—a new development.

To Mr. Stelmaszak, my affable but demanding Calculus teacher, for encouraging students to think about and prepare for the future.

To Mr. Kedigh, for introducing me to programming and computer science.

To Dr. Dave Poplawski, for sponsorship of the student ACM and programming competitions at Michigan Tech.

To Dr. Steve Seidel, for introducing me to the world of research and academe through the MTU UPC seminar.

To Dr. Soner Önder, for teaching me enough of compilers and architecture that I have hardly needed a book or refresher since, a truly amazing skill of a great teacher; I will always remember that I "cannot bribe God."

To Dr. Abdou Youssef, for being an inspiration both in the classroom and out.

To Dr. Poorvi Vora, for your passion for students and teaching.

To Dr. Jonathan Stanton, for introducing me to the world of systems and some of the realities of academic life.

To my advisors, Bhagi and Rahul, for taking me under wing and giving me the freedom to explore.

To Stefan Popoveniuc, for being a great sounding board and working with me on my first paper.

To Eugen Leontie, for being in the trenches with me; our successes have been great and I am glad to have worked with you.

To Joe Zambreno, for useful advice about my career and research. You have helped me to see the world through a different lens.

To Gabe Parmer, for your excitement and input about my research. Our conversations about systems has been great for my intellectual growth.

To Guru, you endorsed my work when I was uncertain in the early stages, which helped me to stay positive and on track.

Friends have a lot to do with how a mind is shaped and grows. Some friends grow and learn with you, teach you, and inspire you to work hard: Dan Mayo, thank you for being such a friend—I am better for having known you.

To Rob Weller, Brandon Wilson, and David Deane, for the boring nights and the exciting, for inviting me out despite my proclivity toward unpredictability and wildness.

To Dan Clark, Justin Ter Avest, and Adam Shirey for 8.31 and all the rest.

To Joe Vaillancourt, for helping to drag me along at times.

To Nick Young and Jeremy Koenen, for broomball, sake, and puyo puyo.

To my friends in grad school who helped lessen some of the burdens of graduate student life, thank you Darby Thompson, Rim Yazigi, Darakshan Mir, Kevin Henry, Amin Teymorian, Liran Ma, Kerry McKay, and Olga Gelbart.

To my lab mates Scotty Smith and James Marshall, for attending more than their share of presentations on my work, and to James for encouraging me to ride my bicycle more, and Scotty for putting up with our bicycle talk.

To the great folks who work with RTEMS, especially Joel Sherrill, Chris Johns, and Sebastian Huber, for the support over the past few years.

To Gary Kreger and Curtis Schoolman, for teaching me that working hard is just working.

To my department and university, and to industry and government for funding my research directly and indirectly through grants, fellowships, and awards. My academic career so far has been supported in part by Hewlett-Packard (through the MTU UPC), the

US National Science Foundation (NSF grants CNS-1117243, CNS-0934725, ITR-025207, CNS-0831149), the Air Force Office of Scientific Research (AFOSR grants FA9550-09-1-0194 and FA955006-1-0152), and the George Washington University (teaching fellowships, travel awards, and summer dissertation fellowship).

To the rest, for surely I missed some, I give thanks.

Surely there must be a less primitive way of making big changes in the [memory] store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself but where to find it.

— John Backus, 1977

Advances in microelectronics have made the realization of “smart” data structures a practical reality.

— Charles Leiserson, 1979

Indeed, I believe that virtually *every* important aspect of programming arises somewhere in the context of sorting or searching!

— Don Knuth

Sisu.

Abstract of Dissertation

Operating System Support for Shared Hardware Data Structures

A fundamental problem in computing is that processors cannot access memory fast enough to stay fully utilized. Architecture features like cache, prefetching, out-of-order execution, and multiprocessing only benefit software with temporal or spatial locality, or instruction-level or task-level parallelism. Software that relies on fine-grained access to data with structural locality, such as pointer-based data structures, derives little benefit from these features. The importance of these data structures motivates a new approach to improve memory performance. A hardware data structure (HWDS) implements a data structure with operations that leverage parallelism and structural locality to reduce data structure access times, but only supports an exclusive data structure small enough to fit the capacity of the HWDS. This thesis proposes operating system (OS) support for HWDSs so that applications can use and share a HWDS even when its capacity is less than the data structure's size.

The priority queue and map data structures demonstrate the appeal of an OS–HWDS union. A GPS benchmark with real-world data executes 24% faster using a HWDS instead of a software data structure, even though the data exceeds the hardware's capacity. Compared to software implementations, a 128-node HWDS achieves over 50% faster mean access time to a 512-node priority queue, and 15% faster mean search time in a 512-node read-mostly map. When sharing a HWDS among four maps of power-of-2 sizes between 64 and 512, a 128-node HWDS achieves 35% faster searches than a splay tree. These performance improvements are made possible by the OS support for HWDSs proposed in this thesis.

Table of Contents

List of Figures

List of Tables

List of Acronyms

API: application programming interface

BST: binary search tree

CAM: content-addressable memory

CSCAA: context switch cost-aware assignment

FIFO: first-in, first-out

HOA: hardware-only assignment

HWDS: hardware data structure

I/O: input/output

ISA: intruction set architecture

LIFO: last-in, first-out

LRU: least recently used

OS: operating system

PAA: priority-aware assignment

RAM: random-access memory

RC: reconfigurable computing

RTEMS: Real-Time Executive for Multiprocessor Systems

RTOS: real-time operating system

SOA: software-only assignment

SPM: scratchpad memory

STL: Standard Template Library

TLB: translation lookaside buffer

TLP: task-level parallelism (or thread-level parallelism)

TM: transactional memory

VMA: virtual memory address

WCET: worst-case execution time

Glossary of Terms

container: An abstract data type in the C++ STL.

exception-based HWDS: A HWDS that permits direct access but raises exceptions when the HWDS cannot satisfy a request. *See also: interposition-based HWDS.*

heap: A data structure containing key-value pairs that orders nodes within a tree according to a rule that a parent node's key is greater than or equal (equivalently less than or equal for a max heap) to its children nodes' keys. *See also: priority queue.*

HWDS assignment: Problem of determining whether a data structure uses a HWDS or a software implementation.

HWDS context: HWDS registers and data associated with a data structure. *See also: HWDS context switch*

HWDS context switch: Saving one HWDS context and restoring another. *See also: HWDS context.*

interposition-based HWDS: A HWDS that is accessed through a software library which avoids making invalid requests to the HWDS by checking every access. *See also: exception-based HWDS.*

locality: The tendency of memory accesses to occur in clusters. *See also: spatial locality, structural locality, temporal locality* .

map: A data structure that contains key-value pairs and supports an efficient mechanism to lookup (search) nodes by key. *Also known as: associative array, dictionary, or search tree.*

multitasking: OS-mediated processor sharing for multiple execution contexts. *See also: scheduler, task, thread.*

node: A storage unit for a data structure comprising one or more data and link (pointer) fields.

priority queue: A data structure that contains key-value pairs sorted by a priority stored in the key.

red-black tree: A balanced tree data structure named for the node coloring rules that ensure a bounded height imbalance. *See also: map*

scheduler: Entity that controls access to hardware resources. Commonly used for sharing processor time or access to devices.

simultaneous multithreading: Hardware-supported processor sharing for multiple execution contexts simultaneously in parallel. *See also: thread, multitasking.*

skip list: A list-of-lists data structure that stores all nodes in the last (bottom) list, and the number of links (height) any given node has is randomized. *See also: map*

spatial locality: Tendency of memory accesses to be located near each other in the memory address space. *See also: locality.*

splay tree: A self-adjusting binary search tree named for the splay operation, which moves recently accessed nodes to the root for faster access. *See also: map.*

split HWDS: HWDS that uses an overflow data structure which ignores the mechanisms of the HWDS. *See also: united HWDS.*

stable: A property of a priority queue or map data structure that dequeues of nodes of the same key is in FIFO order.

structural locality: Tendency of memory accesses to follow an ordered pattern. *See also: locality.*

task: A schedulable software execution context. *Also known as: thread or process.*

temporal locality: Tendency of recent memory accesses to recur. *See also: locality.*

thread: execution context. *See also: simultaneous multithreading, task.*

united HWDS: HWDS that uses an overflow data structure which relies on the HWDS to improve performance. *See also: split HWDS.*

Zipf's distribution: A skewed probability distribution generated with Zipf's law, which states the probability the i'th key out of n keys will be accessed is inversely proportional to i.

Chapter 1 – Introduction

Throughout the history of computing, processors have outperformed main memory [130]. Indeed, the performance gap has steadily increased since the 1980s, leading Wulf and McKee [136] to coin the term *memory wall* to describe the bottleneck caused by the gap. The memory wall arises from processor performance improving faster than memory bandwidth and latency.

One technique to delay the impact of the memory wall is caching. But even with an infinite size cache that (pre)fetches data at full memory bandwidth, the gap between processor speed and bandwidth means cache misses are inevitable—enough data cannot move into the cache fast enough to satisfy the processor. When the cache misses, the memory access time depends on latency to get the first byte, and bandwidth to get the rest. Patterson [99] states that latency lags bandwidth: a historical trend indicates that latency improves slower than bandwidth. Yet latency dominates bandwidth in determining the performance of memory accesses for small sizes, such as a cache line. Poor memory latency means that cache misses become more expensive relative to processor cycle times as technology improves. Ten years ago, a 1 GHz processor with DDR-200 RAM had a memory latency around 52 CPU cycles. Five years ago, a 4 GHz processor with DDR2-800 RAM had a memory latency around 220 CPU cycles.

Meanwhile Moore's law abides: a prediction that a new chip can be produced with double the transistors—potential performance—compared with chips made less than two years prior. As transistor density increases, power and heat dissipation has become a critical factor in chip design and manufacture. The answer from the architecture community has been the chip multiprocessor, or multicore: Excess transistors are devoted either to increased cache or to more processing cores. A fundamental assumption of multicore is that applications can or will exploit sufficient parallelism among multiple cores to achieve speedup. Unfortunately, parallel programming remains hard, despite years of research that has yielded promising technologies such as transactional memory [54] and lock-free data structures [32]. While multicore processors delay the growing gap between latency and performance by processing at lower frequencies, latency still dominates bandwidth, and the memory wall remains.

Scaling the memory wall drives research in both computer architecture and compilers. Computer architects introduced hardware prefetching to reduce miss rates, and techniques to hide cache misses when sufficient work is available—for example non-blocking cache, out-of-order execution, and *simultaneous multithreading*. Compilers play a role in controlling how software accesses the cache and can reduce miss rates using techniques such as software prefetching, instruction reordering, memory compaction, and loop optimizations. Most compiler solutions work well on statically known or easily profiled applications such as software with bounded loops and fixed-size arrays. But many high-level programs are written in terms of data structure (or object) operations and interfaces, and not in terms of loops and arrays.

This thesis improves the state-of-the-art by supporting the use of excess transistors to improve application performance through a fundamental programming construct that spans both processor and memory: the data structure. A hardware data structure (HWDS) is an implementation of a data structure that is supported by hardware mechanisms to improve data structure operations. By organizing the memory hierarchy in terms of data structure operations, instead of cache line fetches, HWDSs permit rethinking how processors access memory. More important, hardware mechanisms exploit parallelism to reduce the algorithmic complexity of data structure operations, which can yield substantial performance benefits compared with software implementations; see Figure 1-1, which shows how

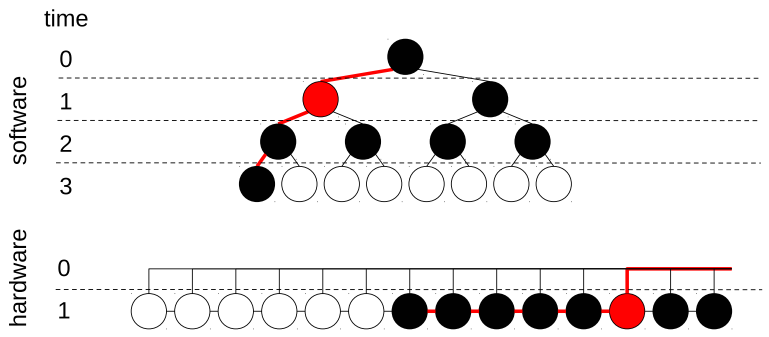
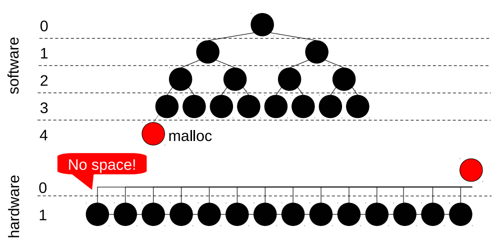


Figure 1-1: The advantage of hardware is parallelism. Here, an insert in a software binary search tree requires traversing at most the entire depth of the tree, whereas hardware can insert in two steps by broadcasting and comparing the new value in parallel.

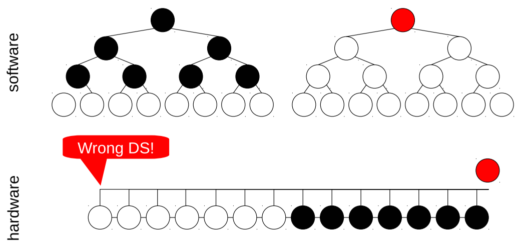
hardware can insert to a sorted structure faster than software because of the advantage of parallel comparisons.

HWDSs are not without disadvantages however, most of which stem from limited hardware resources. Chip space allocated to the HWDS steals from other features such as cache and on-chip communications, so minimizing the HWDS size is important. The main disadvantage of HWDSs is the limited hardware capacity that can be devoted to supporting data structure operations; see Figure 1-2a. Limited hardware capacity precludes using one HWDS for each software data structure, so sharing the HWDS resources in *multitasking* environments is important; see Figure 1-2b, which depicts two data structures attempting to use a HWDS simultaneously. The *HWDS context* is the set of control registers and data belonging to the data structure that is loaded in a HWDS.

Hardware support for specific data structures has been proposed in the past (see Chapter 2), but so far the interface between the HWDS and programmer has been ignored. Most existing HWDSs have limited interactions with operating system (OS) and application software, with much of the prior work allowing only one data structure with a known



(a) A full HWDS cannot accept new nodes.



(b) Data structures cannot share the HWDS.

Figure 1-2: Limited hardware resources create disadvantages for HWDSs.

maximum size (less than HWDS capacity) to use the HWDS; a notable exception is the work of Chandra and Sinnen [27], which is reviewed in Section 2.1 and compared with the approach of this thesis in Section 4.3.2. Sharing a HWDS among arbitrarily-sized data structures requires extra support in both hardware and software.

This thesis shows that OS, application, and HWDS interactions are crucial to realizing efficient HWDSs that arbitrarily-sized data structures can share. Architecture features enable OS and application use of HWDSs. OS support extends the capabilities of HWDSs beyond prior art with support for arbitrary-sized structures and sharing a HWDS among *tasks*. (Throughout this dissertation, *task* denotes a software context and *thread* denotes a hardware context.) HWDSs can also improve the performance of OS data structure operations, and contribute knowledge about task behavior with respect to data structure usage.

Yesterday's data structures were written together with application code. Today's data structures come in optimized, portable, mature libraries. Tomorrow's data structures should ship with the hardware support to use them well. This thesis shows the promise of HWDSs as a new interface between software and memory.

1.1 Impact

Niklaus Wirth wrote that “Algorithms + Data Structures = Programs,” a maxim that has gained strength as software has become more complex and data structures more important. Modern programmers can choose data structures from optimized libraries such as the Standard Template Library (STL) or Boost in C++, and the Java collections framework. These libraries stress both performance and flexibility, but their performance is often limited to an $O(\log n)$ algorithmic factor—and the dynamic nature of these structures lessens the benefits of prefetching and caching. This thesis shows that HWDSs can improve performance by reducing that algorithmic factor to $O(1)$ for common operations in ideal cases, and when the ideal is not met then extra support from the OS helps to maintain performance improvements.

The following examples demonstrate the potential for improvement from data structures implementing the two abstract data types considered in this thesis, the *priority queue* and *map*:

- *Planning algorithms.* Two popular algorithms that use priority queues are Dijkstra’s shortest-path algorithm and the A* planning algorithm. Experiments show that Dijkstra’s algorithm often spends 50–60% of its execution time in the priority queue [81]. Our own experiments on real-world maps taken from the DIMACS shortest path implementation challenge benchmarks [26] show the benchmark spends up to 29% of its time inside the priority queue.

- *Image analysis.* The grey-weighted distance transform on 3D images uses a software priority queue [82]. Measurements show the priority queue accounts for over 30% of the application’s execution time; see Section 6.4.

- *Discrete event simulation.* A priority queue organizes pending events in a discrete-event simulation (such as a queueing network or integrated circuit simulation), and has been a popular test case for priority queue implementations [61, 105]. Such simulations spend up to 40% of execution time managing the queue [105].

- *Fine-grained multitasking.* Carbon [73] uses hardware queues to improve fine-grained multitasking for Recognition, Mining, and Synthesis. Compared to software approaches, Carbon can achieve 68% faster execution time for loop-level parallelism, and 109% for task-level parallelism.

- *Real-time task scheduling.* In prior work, I have shown that a hardware priority queue reduces scheduling overheads and improves predictability [16]; others have shown that a hardware priority queue can reduce task *scheduler* overhead from 18% in software to less than 0.5% [72].

- *Web browsers.* The Chromium web browser makes extensive use of the C++ STL map container, which often is implemented as a red-black tree. Profiling (see Appendix A) of this code shows that—even for a short session of starting, loading a blank page, and stopping—Chromium creates 1907 maps and executes 49,483 *find* operations that consume 436,758,391 cycles of map execution time, or approximately 12% of overall execution time.

- *Programming languages.* Interpreted languages need to look up strings frequently, as do systems that monitor memory accesses. For example, Akritidis et al. [7] use a splay tree—a self-adjusting binary search tree (BST)—referent object checker and evaluated it on the Olden and SPECINT 2000 benchmarks—for Olden the time overhead of using the checker was 30% on average (excluding two benchmarks); for SPECINT 2000 the overhead was on average 900% and exceeded 100% for all benchmarks.

- *OS search trees.* Pfaff [100] evaluates implementations of BSTs—including random BSTs, self-balancing BSTs (AVL and red-black), and splay trees—in the context of systems usage. The systems applications used to evaluate the BSTs are virtual memory address (VMA) mapping in Linux, IP peer caching, and index cross-reference collation. With real-world data, a splay tree implementation of VMA mapping improves performance of Mozilla, VMware, and Squid test sets by 23% to 40%. Other uses of balanced search trees in Linux include: input/output (I/O) schedulers, optical device driver, high-resolution timers, ext3 filesystem directory entries, and cryptographic keys [1].

- *Key-value stores* Key-value stores implement straightforward searching with keys that are often either strings or integers. Search benchmarks model the application processing of key-value stores; OS processing time of key-value stores can be substantial—when requests are small memcached spends up to 80% of its time in OS code primarily for network packet processing [20].

These applications are just a sample of the uses for priority queues and maps. OS support for HWDS use in these applications can eliminate much of the time spent processing data structure operations.

1.2 Overview

In using a data structure, an application "reads" (searches or iterates) and "writes" (inserts or removes) *nodes*. A data structure's read/write operations abstract the lower level load/store operations that comprise a processor's interface to memory. By supporting the high-level abstraction of data structure operations, HWDSs enable applications to extract fine-grained parallelism from their data structures.

```
void bubble_up(int i) {
  while (i > 1 && heap[i]->key <
  heap[PARENT(i)]->key) {
    swap_entries (i, PARENT(i));
    i = PARENT(i);
  }
}
void heap_insert(int key,int val) {
  int s = ++heap_current_size;
  heap[s] = alloc_init_node(keyval);
  bubble_up(s);
}
```

(a) Insertion code for a software binary heap.

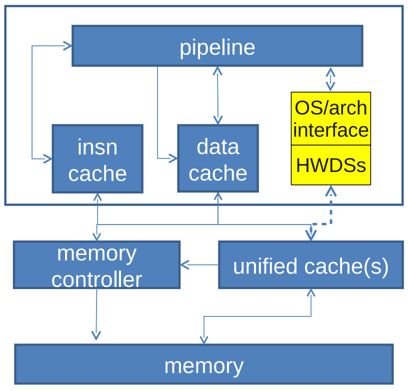
```
void pq_insert( int id, int val, int key){
  HWDS_INSERT(id, key, val);
}
```

(b) Insertion code for a priority queue HWDS.

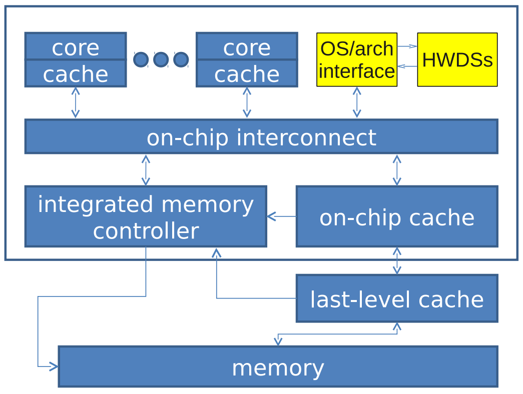
Figure 1-3: Program code changes when using a HWDS.

Figures 1-3a and 1-3b demonstrate the expressive power of a HWDS abstraction with the insert code of a software priority queue implemented as a binary heap, and the insert code of a priority queue using a HWDS respectively.

Figure 1-4a shows how a HWDS can fit with other computer hardware in a uniprocessor setting; multicore chips introduce complications for sharing and communication, and one possible configuration is shown in Figure 1-4b. Design space exploration for both uni- and multi-processing with HWDSs is interesting future work.



(a) Computer organization.



(b) Multicore computer organization.

Figure 1-4: HWDS architecture overview.

This thesis makes it possible to use a HWDS even when the application's data needs exceed the HWDS capacity, or when multiple data structures attempt to share the HWDS concurrently. I demonstrate the benefit of OS support for HWDSs with use cases and synthetic benchmarks that are executed using cycle-accurate simulation.

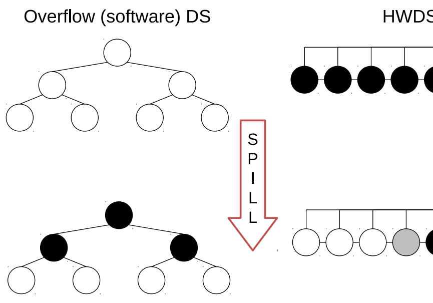
1.2.1 Overflow handling

Generic applications require support for data structures of arbitrary size. Since hardware has a fixed capacity, arbitrarily large data sets eventually will cause overflow. A HWDS is like a write-back cache: it must save dirty nodes to backing storage or else the updated data would be lost. This is in opposition to a write-through or read-only cache, which can handle overflow by simply removing nodes from the hardware unit's storage because the backing storage already contains the up-to-date node's data.

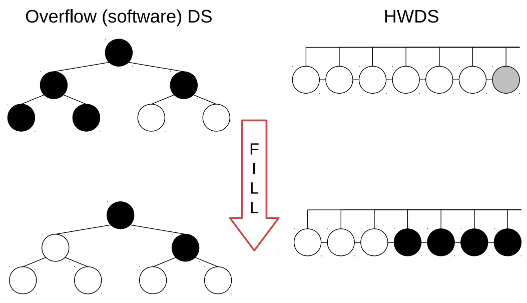
The specifics of overflow handling depends on the implementation of the HWDS, but the general concept is universal. To deal with overflow, HWDS control logic and software (for example, the OS) spill data out of the HWDS and into an overflow data structure in secondary storage (main memory); see Figure 1-5a. Conversely, control logic and software fill data from the overflow data structure when the HWDS needs to access nodes that it previously spilled; see Figure 1-5b. Section 3.1 describes HWDS overflow handling in greater detail.

1.2.2 Sharing HWDS resources: HWDS assignment

Multiple data structures might share a HWDS, for example when two applications execute concurrently and use the hardware for different data structures. Sharing is a traditional OS problem of how to manage contention for a limited hardware resource: The usual solution is scheduling. This thesis turns the sharing problem into that of *HWDS assignment*, which



(a) Spilling to handle overflow.



(b) Filling to handle underflow.

Figure 1-5: Handling limited hardware capacity with HWDSs.

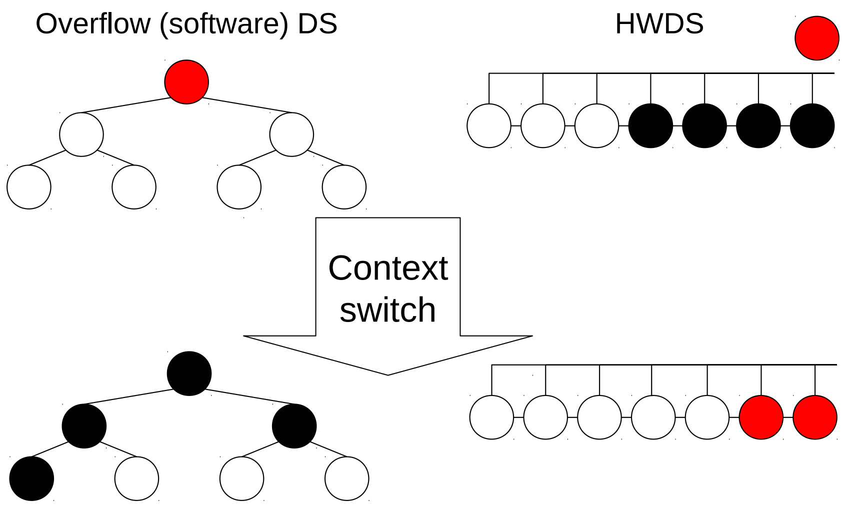


Figure 1-6: HWDS sharing with a HWDS context switch.

is the problem of determining whether a data structure uses a HWDS or a software-only implementation. When two data structures do share a HWDS, the OS supports the HWDS with a *HWDS context switch*—spilling the nodes for the current HWDS context and filling nodes for the requested data structure; see Figure 1-6. Section 3.2 further illuminates the problem of sharing HWDS resources and its solution, HWDS assignment.

1.3 Contributions

This thesis explores the hardware-software interface of HWDSs with a holistic approach that has many contributions including:

- **Operation-level interface for applications to use HWDSs.** An interface between HWDSs and software gives applications access to HWDS resources and improves program performance. The programming interface is at the level of data structure operations, and the implementation is at the instruction set architecture (ISA) level so that future improvements in the hardware microarchitecture do not affect the interface.

- **Effective use of parallelism compared to conventional architectures.** Explicitly parallel architectures require a programmer to partition and synchronize shared data accesses. HWDSs use implicit parallelism to achieve high-performance parallel computing without burdening the programmer with consistency and tasking models. Implicit parallelism improves software performance at little cost to the programmer.

- **Spilling HWDS overflow data.** Hardware and software work together to support large data structures that overflow hardware capacity. Although some performance is lost, the HWDS approach remains competitive with software-only solutions. Compared to software implementations, a 128-node HWDS achieves over 50% faster mean access time to a 512-node priority queue, and 15% faster mean search time in a 512-node read-mostly map.

- **HWDS Assignment for sharing a HWDS.** HWDS assignment is supported by the OS to share and restrict available HWDS resources among multiple data structures. When sharing a HWDS among four maps of power-of-2 sizes between 64 and 512, a 128-node HWDS achieves 35% faster searches than a *splay tree*. Eviction of oversized HWDSs enables the OS to make dynamic assignment decisions to limit performance loss; when a 128-node HWDS is used for a 512-node map that is updated and searched, an eviction policy yields 16% performance loss, but performance loss without eviction is 64%. Prior art does not offer any solutions for HWDS assignment, so these performance improvements are made possible solely by the OS support for HWDSs proposed in this thesis.

- **Support for many kinds of data structures.** The priority queue and map are examples of HWDSs that improve the performance of sorting and searching, two fundamental problems in computing. The policies and solutions of this thesis apply to both kinds of data structures, and future work can investigate others such as string-based or hashing structures.

- **Increased real-time schedulability.** HWDSs can benefit real-time systems by reducing worst-case execution times (WCETs) even when multiple data structures share a HWDS or when data structure sizes exceed HWDS capacity.

- **Evaluation with cycle-accurate timing, real systems, and real-world data.** Real applications and synthetic benchmarks validate the HWDS approach using cycle-accurate fully-functional simulation. OS support is designed and implemented in the Real-Time Executive for Multiprocessor Systems (RTEMS) real-time operating system (RTOS), so real OS overheads are included in the experiments. The simulator executes HWDS operations and accounts for operation latency as part of the cycle time. Experiments are conducted using applications and microbenchmarks that use data structures with both software and HWDS implementations.

With respect to prior art, an experiment using a GPS benchmark with real-world data is conducted that compares overflow handling with the exception-based *united HWDS* proposed by this thesis with the interposition-based *split HWDS* proposed by others [27]; see section 4.3.2. When using the united HWDS, the benchmark executes 24% faster than when using a software implementation, even though the data structure size exceeds the hardware's capacity. The benchmark using the split HWDS *never does better than software* in the presence of overflow.

The OS support for HWDSs presented in this thesis bears some resemblance to policies and mechanisms for cache and translation lookaside buffer (TLB) management, but the *structural locality*, operation diversity, and design and implementation multiplicity of HWDSs demand new solutions. Memory cache is a reflection of a flat array of storage, and leverages the independence between cache lines for fast, effective fetching and replacing. A HWDS has connections between nodes that must be preserved, which would require complex hardware to implement structure-preserving overflow. HWDSs support common data structure operations that encode high-level abstractions in low-level mechanisms, whereas cache and TLB are limited to the load/store memory interface. Extant solutions to hardware overflow and sharing that rely on hardware mediation are not useful across multiple kinds of HWDSs, and hardware management for any given HWDS implementation would drive up its cost and complexity in terms of both development and hardware resources. The structural locality, operational richness, and design diversity motivate software management of HWDSs. This thesis shows that software—more flexible, fixable, and forward-compatible than hardware—can manage HWDSs efficiently to provide performance gains for applications and systems software.

1.4 Scope

Investigation of HWDSs is an open-ended area of research. Limits on the scope of this thesis delineate what is and is not investigated.

This thesis investigates: architectural support for HWDSs with Simics/GEMS, OS support with RTEMS in a uniprocessor setting, representative data structures (priority queue and map) and applications, HWDSs in real-time systems, and the performance of HWDSs versus software-only solutions.

This thesis does not investigate: real hardware or general purpose OS (e.g. Linux) implementations, design space exploration for HWDS interfaces or implementations, compiler support for HWDS, sharing a HWDS among multiple tasks with a single data structure, OS optimizations that use the knowledge about applications gleaned from HWDS behavior, multiprocessor architectures, and metrics related to power, reliability, or usability. All of these areas are possible directions for future work.

1.5 Outline

This thesis is organized as follows. Chapter 2 reviews the related work in the field. Chapter 3 describes the generic OS support for HWDSs necessary for overflow handling and HWDS assignment. Chapter 4 describes an example of a HWDS that implements a priority queue, refines the generic overflow handling support, and presents experimental results that demonstrate the performance of overflow handling and HWDS assignment for two important priority queue applications: discrete event simulation and path planning. Chapter 5 proposes a HWDS implementation of a map for efficient searching, and presents experimental results from a synthetic search benchmark. Chapter 6 shows how real-time systems can use HWDSs to improve the schedulability of task sets by reducing WCETs; I evaluate four HWDS assignment algorithms using experiments and benchmarks modeled from real-world applications. Chapter 7 identifies possibilities for future work and concludes.

Chapter 2 – Literature Review

The work most closely related to this thesis are in the areas of design of HWDSs, hardware support for fine-grained parallelism, shipping code to data, linked prefetching, object-based systems, and transactional memory. The following reviews each of these in turn.

2.1 Design and Implementation of HWDSs

2.1.1 HWDSs for network routing

Hardware support for scheduling has been an area of interest in the queuing hardware of packet-switched networks. Moon et al. [87] compare four approaches to hardware priority queues for high-speed networks and introduce an approach that melds two of the previous solutions. Kim and Shin [65] describe an architecture for EDF scheduling for ATM switch networks and introduce deadline folding to circumvent limitations in the range of priority values. Bhagwan and Lin [14] introduce a heap-based hardware priority queue with pipelined stages of the enqueue and dequeue operations. Morton et al. [89] describe a hardware priority queue that does not require hardware comparators.

How this thesis differs Although packet-switched routers can benefit from hardware priority queues, software has no interface to access the priority queues—they are only useful for sorting network packets. This study enables software to use the priority queues by exposing an interface to the hardware so that software can benefit from the hardware acceleration while remaining flexible to implement different algorithms using functional memory.

2.1.2 HWDSs for real-time scheduling

Approaches for hardware-based packet scheduling have been extended for task scheduling in RTOSs. The goals of hardware support for real-time scheduling are to minimize scheduling latency and provide highly predictable multiprocessing. The Spring Scheduling Coprocessor (SSCoP) [24] is one of the first examples of a hardware task scheduler and introduces simple queues for the set of scheduled tasks. Others have implemented hardware scheduling using some form of custom logic and a hardware priority queue [108, 71, 69, 16, 72, 115].

How this thesis differs In contrast to the prior work, which focuses on hardware support for a single fixed-size priority queue, this thesis allows arbitrarily-large priority queues to share a hardware priority queue.

2.1.3 HWDSs for reconfigurable computing with Java

Chandra and Sinnen [27] investigate HWDSs in the context of integrating a high-level language, Java, with reconfigurable computing. In addition to the usual priority queue operations, the authors investigate how to increase the queue length, use non-integer priority values, and add new operations.

How this thesis differs Chandra and Sinnen do not consider how HWDSs are shared and scheduled among multiple consumers. Their approach, a split *interposition-based HWDS*, does not handle overflow well; see Section 4.3.2.

2.1.4 Systolic Priority Queues

Leiserson [77] describes systolic HWDS implementations including priority queue, multi-queue, and tree. He suggests that overflow be handled by the OS, and that pairing an insert with an extract can handle refilling the HWDS.

How this thesis differs Leiserson focuses on the hardware design of systolic HWDSs with only cursory examination given to the software-side of the HWDS-OS equation. This thesis demonstrates that intelligent software support is necessary to achieve good performance from HWDSs in the presence of overflow and sharing.

2.1.5 Abstract Datatype Processors

Kim [67] and Wu et al. [134] share the vision of raising the abstraction of hardware to that of software; their work proposes and evaluates abstract datatype processors, which accelerate data types with mechanisms and performance similar to HWDSs. Abstract datatype instructions can reduce instruction fetch times by 21–48% and data read/write times by 22–40%. The datatypes they investigated are the sparse vector and hash table, and hardware support is modeled with a content-addressable memory (CAM).

How this thesis differs Abstract datatype instructions currently ignore capacity and sharing problems, but the similarity between these instructions and HWDSs indicates similar problems exist due to hardware size limitations.

2.1.6 Content-addressable memory (CAM)

Hardware can search small sets of records with numerical keys efficiently with a CAM. Ternary CAMs [97] can implement approximate search for some applications, such as longest prefix matching.

A common use for CAM in modern computing is as a read-only cache for page tables—the virtual-to-physical address translation map that underlies page-based virtual memory systems. This cache is called the TLB, and its purpose is to cache translations for fast lookup. Tagged TLBs permit cached entries from multiple page tables to share the TLB. TLB overflow is handled by dropping entries; since the TLB is a read-only cache, the backing data remains in memory. However if the page table is modified, the TLB needs to be refreshed or invalidated.

How this thesis differs CAMs do not permit searching with arbitrary-sized or multiple data sets because of limited hardware capacity, but the solutions posed in this thesis may be used with CAMs to implement a map HWDS.

The primary difference between the page table-TLB and the HWDSs employed in this thesis is that the TLB acts as a read-only cache for the page table, whereas this thesis uses HWDSs like a write-through cache for the overflow data structure. Although subtle, this difference is important. Other differences include: a TLB does not export a SEARCH function; a task or process only gets to use one page table at a time; TLBs do not in general support arbitrary search keys—the address translation relies on the size of pages in the page table to divide the search space.

2.1.7 Scratchpad memory (SPM)

An alternative to caching in the embedded domain is a scratchpad memory (SPM) [101]. SPMs can provide predictable access times and software control over code [133] and data [125]. SPMs are software-managed: applications and compilers control the data (and code) residing in the SPM. Co-mingling SPMs with custom hardware can provide further benefits such as intelligent object-based allocation [129, 128].

How this thesis differs Software that uses a SPM still executes serially to access data structures. HWDSs execute in parallel and require different management than scratchpads because of the increased hardware complexity in HWDS logic. Combining the two approaches to use a HWDS with a SPM as the backing store may be useful for overflow handling.

2.1.8 Reconfigurable computing data structures

A seminal paper in reconfigurable computing (RC) design by Dehon et al. [40] proposes classes of design pattern for RC. One of these design pattern classes is the *Value-Added Memory Patterns* which includes CAMs, priority queues, and other data structures. Some of the other data structures implemented in RC logic include graphs [85, 41] and trees [117]. These data structure implementations can be reused most easily in a HWDS framework that executes as a co-processor.

How this thesis differs Existing RC data structures do not support general applications because sharing and overflow are not addressed.

2.1.9 String matching

Modern applications increasingly rely on text processing—for example parsing web documents, string search, and regular expression matching—that benefits from hardware support for string matching [25]; so do network appliances for deep packet inspection in intrusion detection [30, 118, 63, 139, 60].

How this thesis differs String and regular expression matching architectures implement HWDSs for specialized string-based applications. Future work can make use of these HWDSs and improve their generality across application domains by employing the results of this thesis.

2.2 Fine-grained Parallelism

If a programmer decomposes a program into small independent tasks then the program has more potential parallelism and, by Amdahl's law [10], greater speedup. Thus, multicore platforms should support fine-grained task-level parallelism (or thread-level parallelism) (TLP) for greater speedup. The challenge for fine-grained TLP is to maintain overheads proportional to task sizes and to avoid solutions that degrade performance, for example by destroying locality. Improving TLP performance for current and next generation processors shares common ground with HWDSs, both in motivation and solution methods.

2.2.1 Carbon

Kumar et al. introduce Carbon [73], hardware acceleration for multicore task scheduling with task last-in, first-outs (LIFOs), prefetchers, and work stealing in hardware to support fine-grained TLP. Carbon exposes a task queue application programming interface (API) in the form of intruction set architecture (ISA) extensions, so it is similar to the HWDS paradigm.

How this thesis differs In Carbon, the queues are used specifically for task scheduling, which means that applications only benefit if Carbon extracts sufficient fine-grained TLP. Carbon provides no benefit to serial workloads and requires small task sizes to see improvement over software scheduling. A HWDS configured as a LIFO would be similar to the single core configuration of Carbon.

2.2.2 Ne-XVP

A research project at NXP Semiconductors (formerly Phillips Semiconductors), the Ne-XVP architecture aims to provide an efficient multimedia processor platform. Three specific aspects of the Ne-XVP are relevant to this study: the Task Scheduling Unit) [56], Task Management Unit) [113], and Hardware Task Scheduler [8]. Unlike with Carbon and HWDS, the hardware queues in Ne-XVP are not exposed at an API or ISA level.

How this thesis differs As in Carbon, the scheduling policy is inflexible and programs that lack TLP cannot improve from the extra hardware support. Our project allows programs to improve serial performance bottlenecks by taking advantage of parallelism in data structures.

2.2.3 Asynchronous Direct Messages

Sanchez et al. [109] introduce asynchronous direct messages (ADM) to provide message passing akin to interprocessor interrupts but avoiding the cache hierarchy. The authors implement work-stealing scheduling algorithms for multicore platforms in the context of fine-grained parallel workloads using ADM. Task queues are maintained in software, so that ADM is the only hardware component of the task scheduler. New privileged software handles the receive buffer overflow and underflow conditions. Privileged software also is responsible for mapping each scheduled task to a specific core for translating destination task IDs when routing messages.

How this thesis differs Asynchronous direct messages attack the communication bottlenecks between tasks in a multicore platform, whereas this thesis focuses on the bottleneck of serial memory accesses during data structure operations.

2.2.4 HAQu

Lee et al. [76] propose a hardware accelerated queue (HAQu, pronounced "haiku") that accelerates software queues for multicore platforms. Unlike the work reviewed so far, HAQu does not use a hardware queue; instead HAQu implements queuing through an application's address space. Hardware buffers queue operations through a unit that complements each core's pipeline.

How this thesis differs Implementing fast data structures through an application's address space is an interesting idea, but the restrictions on use (single producer-consumer pairs, memory fences) may be trouble for complex data structures. Because HAQu does not leverage hardware parallelism, it cannot achieve the speedup possible with a true HWDS. Future work may consider how HWDS can provide isolation of producers and consumers while maintaining memory consistency in hardware and virtualization using the address space write-through proposed by HAQu.

2.2.5 Loop accelerators

An approach for exploiting certain kinds of loop-level parallelism is a loop accelerator. Loop accelerators typically sit on the system bus and directly access memory. Often they are customized for a particular loop or a limited range of loop bodies. Loop accelerators excel over general purpose processors by exploiting loops with simple control flow, cyclically repeated instruction streams, decoupled memory accesses and computations, and domain-aware customizations of the processing units (functional units, interconnect, register files) [33]. The same reasons that loop accelerators are advantageous to use prevent them from being useful for complex or linked data structures. Branches within iteration cannot be speculated easily within a loop accelerator, so structures having branch points such as trees will not be supported.

How this thesis differs Linked structures are hard to accelerate in a loop accelerator because the address generation hardware is unable to use simple computations to fetch the required memory for a loop body. Random access also implies data-dependent address calculations, so certain array-like structures are not suitable for loop accelerators. These restrictions prevent comparison of loop accelerators with HWDSs because the two approaches target distinct workloads. Future work can consider approaches that combine loop acceleration with HWDS support for linked data structures.

2.2.6 Scalable Cores

Hill and Marty [55] argue that architecture research should pursue methods that provide the ability to combine cores dynamically to boost the performance of sequential code—Gibson calls such processors *scalable cores* [51]. CoreFusion [58], TRIPS [110], Composable Lightweight Processors [66], WiDGET [127], and ForwardFlow [52] are scalable core architectures. Scalable cores adapt dynamically to the needs of software so that TLP is exploited when sufficient parallelism exists, while sequential workloads benefit from aggregations of execution units.

How this thesis differs Scalable cores take an execution-oriented view toward performance and choose between offering ILP or TLP. Like scalable cores, this thesis improves the performance of (data structure) code that is hard to parallelize at a task granularity; the difference is that a data-oriented view provides speedup to workloads that may not benefit from either ILP or TLP because the primary bottleneck is memory.

2.3 Shipping Code to Data

2.3.1 Data structure co-processing

Loew et al. [81] introduce data structure co-processing as a hardware-software approach for accelerating data structure operations. This approach is a model of computation that offloads data structure operations to a separate hardware thread or core. The main drawback of the model is that the offloading suffers poor performance due to synchronization and communication between application and data structure threads.

How this thesis differs This thesis couples HWDSs with OS support for applications. HWDS could improve data structure co-processing through core specialization.

2.3.2 Processor-in-memory

Shrinking memory bandwidth with respect to processsor speed motivates intelligent memory (processor-in-memory), for example the IRAM project [98] and Active Pages [95]. An intelligent memory architecture embeds some processing units close to memory, that is, on the same chip as the memory modules. The processing units enable computations that can use memory at a higher bandwidth than a traditional CPU over a memory bus. Other processor-in-memory projects include [98, 44, 59, 47, 70, 21, 31, 119, 46, 140]. OS support such as that of ActiveOS [94] for Active Page enables intelligent memory for multiprocess environments.

How this thesis differs HWDSs differ from intelligent memory by taking advantage of parallelism within structured data; the two approaches could be used together with a HWDS implementing an intelligent memory processing unit. This study in particular focuses on the OS policies and support needed to make HWDSs work with general-purpose applications.

2.3.3 Processor-in-disk

Disk I/O suffers similar latency problems as memory, and improvements in disk I/O performance would benefit applications such as databases, web transaction processing, data mining, and multimedia. Early work in database processors [114, 96, 79, 111] reduce the costs of relational database operations by tailoring circuits to access data independently from main processors. Database processors were abandoned due to inflexibility and problems with backward compatibility [19], but active disks and related approaches [6, 104, 64] generalize database processors to improve general-purpose disk I/O by shifting general-purpose processors into disk controller interfaces. The notion of shifting processing code to disks leads to semantically-smart disks that integrate disk I/O with knowledgeable filesystems and applications [112].

How this thesis differs As with the processor-in-memory work, this thesis focuses on the OS policies and support for sharing and handling overflow. Furthermore, the implication of intelligent disks is that either applications provide disk processing code, or disk devices are application-aware. With HWDSs, the abstraction of a data structure precludes such tight integration between hardware and software.

2.4 Linked Prefetching

Prefetching is a known commodity in modern computer architecture. But just as well-known is that prefetching works well in structures that exhibit high *spatial locality*: iterating through dense arrays being the best case. For non-local accesses, such as those seen in linked data structures, traditional prefetchers can actually degrade performance due to unnecessary fetches. Prefetching of linked data structures is a challenging research area with interesting solutions, including correlation-based prefetching [62], pointer prefetching [107, 126, 23], content-directed prefetching [36, 45], and push prefetching [137, 138]. A novel solution also combines linked prefetching with intelligent memory in which a programmable unit traverses data structures in memory and feeds the processor with prefetch data [57].

How this thesis differs Unlike linked prefetchers, the HWDS implicitly knows the structure of data so there is no need for logic to look-ahead and fetch from memory. Linked prefetchers offer one side of a coin—reduce average memory access times for linked data structures—with a HWDS on the other side of the coin: reduce data structure processing times through fine-grained parallelism. Combining the two approaches would be interesting; perhaps a linked prefetcher could implement the overflow/underflow handler of a HWDS independently of software.

2.5 Capability- and Object-based Systems

From the mid 70s through the late 80s, computer architects sought to support capabilities [42, 78] and object representations directly in hardware [135, 34, 12, 90, 38]. An infamous commercial system is the Intel iAPX 432, which featured capabilities, object addressing, garbage collection, interprocess communication, multitasking, and multiprocessing [93, 37, 78]; the iAPX 432 design failed due to performance problems [35].

How this thesis differs Every language can implement an object representation, so direct hardware support for objects is inflexible and non-portable, and OS modifications are intrusive—especially for hardware capabilities—and complex. HWDSs implement an abstraction that permits simple hardware and modular OS support. Future work can extend this thesis to show how to support objects in a manner consistent with HWDSs.

2.6 Transactional Memory

Multicore places extra pressure on memory: programs share data and execute in parallel contending for shared memory. Traditional solutions for contention—namely locking—may not scale well to multicore systems. An alternative solution is transactional memory (TM) [54]—in the spirit of database transactions—that provides an all-or-none atomicity for memory accesses.

Hardware support for TM alleviates performance concerns, and both hardware-only and hybrid hardware/software TM systems have been proposed and produced [88, 84, 18, 43, 39]. Some of the challenges with TM systems is integration with the OS, for example how to use transactions in the presence of system calls and I/O [106, 124].

How this thesis differs HWDSs provide benefits to serial code through fine-grained parallelism within data structure operations, an advantage that TM cannot produce; TM relies on the availability of task parallelism and multicore processing. Future work can investigate how HWDSs in a multicore platform can provide HWDS-mediated sharing and compare that with TM.

2.7 Summary of Related Work

None of the prior art approaches the problem of memory limiting system performance from a HWDS point-of-view. Implementation similarities between HWDSs and other systems abound, and I have reviewed those which are most similar. This thesis shows that OS support elevates HWDSs to improve applications in multitasking environments. The related work suggest other areas that HWDSs may benefit, such as reconfigurable computing or work-offloading in a multicore platform.

Chapter 3 – OS Support for HWDSs: Generalities

Software support can help circumvent the size and sharing limitations of hardware so that applications can benefit from HWDSs. This chapter describes the generic aspects of such support, and subsequent chapters describe aspects that are specific to the priority queue and map HWDSs.

3.1 Overflow Handling

This study, inspired by work in fine-grained task-level parallelism [73, 109], adopts an *exception-based HWDS* approach, as opposed to an interposition-based HWDS [27] which avoids exceptions by checking (with software) before every HWDS access. The HWDS generates an overflow exception when the size of the data structure exceeds the capacity of the hardware. An overflow exception handler then processes the exception by spilling nodes from the HWDS. When the used capacity of the HWDS falls below a programmable threshold—and there are spilled nodes—control logic raises an underflow exception. The underflow handler fills the HWDS from the overflow data structure.

Spilling causes a problem for operations that target spilled nodes: software must implement the operation on the nodes in the spill area. When an operation fails while using an exception-based HWDS, control logic raises a failover exception and the exception handler emulates the operation on the nodes in the overflow data structure. (Interposition-based approaches must determine whether an operation should target the spilled nodes or the HWDS.)

Overflow handling introduces the following HWDS instructions:

- `GET-CONTEXT`: queries the context of the HWDS to determine the cause of an exception and the interrupted instruction.

- `SPILL`: Moves a node from the HWDS to backing storage.
- `FILL`: Moves a node from backing storage to the HWDS.

A HWDS implements `SPILL` with the data structure's delete operation, and `FILL` as an insert operation, where the node to delete or insert is chosen according to a HWDS-specific policy. Sections 4.2 and 5.2.2 describe these policies for hardware priority queues and hardware maps, respectively. `GET-CONTEXT` can be implemented in a control unit alongside the HWDS that stores the most recent HWDS operation and its arguments.

Exceptions allow software to be oblivious to the HWDS capacity, but they induce overhead that reduces the throughput and predictability of applications. The cost imposed by overflow handling depends on the implementation of the overflow data structure, frequency of overflow/underflow/failover, and the cost of executing the exception handler. Experimental evaluations in subsequent chapters of this thesis quantify the costs of overflow handling.

3.2 HWDS Assignment

Sharing the HWDSs adds complexity to both hardware and support software. The main addition is that the hardware needs to distinguish data structures; in prior work, there was a one-to-one mapping between data structure and HWDS. Loosening that mapping to many-to-one introduces the problem that the HWDS must have some way to separate or distinguish data structures and their operations. As with other facets of HWDS design, more than one solution exists for this problem. The solution I adopt is to add an identifier to every instruction that accesses the HWDS and for the hardware to track which data structure currently is in use. Exception handlers use the identifiers to store overflow in separate data structures. I chose this approach because the hardware cost is small (an extra register and some comparators) while supporting a wide range of policies for how HWDSs are shared. The main drawback is that each data structure must have a unique identifier.

HWDS assignment introduces the following instructions:

- `SAVE-CONTEXT`: save the data from a HWDS to backing storage and make that HWDS available for use
- `RESTORE-CONTEXT`: assign the HWDS and (optionally) restore data

A HWDS context switch is a `SAVE-CONTEXT` followed by a `RESTORE-CONTEXT`. As the utilized capacity of a HWDS increases, the cost to `SAVE-CONTEXT` also goes up. In Section 6.4, I evaluate assignment algorithms that can limit the usable size of a HWDS in order to limit the cost of the context switch.

HWDS assignment can be solved statically or dynamically. Static assignment determines offline which data structures are assigned to use HWDS resources and at runtime a HWDS context switch swaps one assigned data structure for another. Dynamic assignment permits the OS to make assignment decisions online. Some mechanisms for dynamic assignment are (1) permitting data structure operations to proceed without hardware support (assignment to a software implementation), (2) saving the context of the currently in-use HWDS and restoring the context of the requested data structure, or (3) suspend the task making the request until a HWDS becomes available. With (1), every data structure operation raises an exception that emulates the operation in software—a prohibitively expensive solution. (Interposition-based approaches can implement (1) without such expense.) Mechanism (2) has the drawback that it can lead to a problem analogous to thrashing; in the worst case, every access to a HWDS could cause a `SAVE-CONTEXT`. A concern with (3) is starvation. This thesis presents and evaluates static assignment algorithms and dynamic assignment using mechanisms (1) and (2); future work can evaluate policies and algorithms for HWDS assignment more thoroughly.

If software attempts to access a data structure that is not presently in the HWDS context, then control logic triggers an exception. The OS can assign the data structure to an available HWDS, save the context of a currently used HWDS and assign it to the data structure, or assign the data structure to use software only.

3.3 Experimental Infrastructure

I implemented HWDSs in the Simics/GEMS simulator [83]—a functionally correct, cycle-accurate full system simulator for an out-of-order architecture (based on the Alpha architecture) that executes the SPARC v9 instruction set. The architectural parameters I chose are representative of an embedded system: 75 MHz CPU, 80 cycle memory latency, and a 4-issue 5-stage pipeline. The implementation extends the SPARC instruction set to use a reserved opcode for HWDS instructions, which are executed with a new functional unit. This functional unit operates atomically and non-speculatively. Although the HWDS can achieve single-cycle latencies for priority queue operations, restricting the unit to be atomic and non-speculative increases the latency to around 12 cycles for the simulated architectural parameters.

I modified RTEMS [91] to provide OS support for HWDSs. OS modifications include HWDS exception handling, overflow data structure implementations, task scheduling algorithm implementations, a rudimentary HWDS interposition library, and macros to emit HWDS instructions. I also modified the GCC compiler to support the HWDS instructions, although presently the only way to emit these instructions is with hand-written assembly.

Chapter 4 – Priority Queue HWDS

This chapter shows how the priority queue HWDS can be supported by the OS to support applications that may require overflow handling for large data sets or concurrent access to the HWDS resources. HWDSs can be effectively used in multitasking environments when the hardware is managed properly. Intuitive solutions for sharing and overflow do not achieve adequate performance; in the presence of overflow, simply using a well-known and efficient overflow data structure leads to worse performance than using a software-only data structure implementation.

4.1 Priority Queue: an Example HWDS

A priority queue is a data structure that contains key-value pairs where the key is a priority upon which the structure is sorted. Usual operations on a priority queue are:

- PEEK [first, top]: returns the highest priority node
- ENQUEUE [insert, push]: adds a new node
- DEQUEUE [delete-min, pop]: removes and returns the highest priority node
- CHANGE-KEY [decrease-key]: modifies a node's key (priority)
- EXTRACT [delete]: removes a given node regardless of priority
- MERGE [meld]: combines two priority queues into one

A priority queue is *stable* if nodes with the same priority are dequeued in first-in, first-out (FIFO) order. The importance of priority queues to application performance can be seen in the examples of Section 1.1.

4.1.1 Software priority queues

As many software priority queue implementations exist as sorting algorithms: Any sorting algorithm can implement a priority queue [123]. For example, insert-sort implements a priority queue with a linked list, heap-sort implements a priority queue with a *heap*, and tree-sort implements a priority queue with a BST. A traditional priority queue implementation uses a heap; an implicit heap, which stores a binary heap as an array, is a common implementation. Variants of the heap include the binary heap, implicit heap, leftist tree, binomial queue (binomial heap), pagoda, skew heap, Fibonacci heap [49], pairing heap [48], Brodal queue [22], and soft heap [28]. BSTs can implement priority queues by keeping track of the extreme (min and max) values in the tree. Common BST implementations of a priority queue use a *red-black tree* or a splay tree. An advantage of a BST over a heap is that the BST can more readily handle duplicate keys (tied priority).

In 1986 Jones concluded "[i]mplicit heaps are among the worst choices for queues smaller than 20 nodes–and consistently worse than other priority-queue implementations" [61]. But in 1996 LaMarca and Ladner [74] stated this rebuttal:

> [T]he low memory overhead of implicit heaps makes them an excellent choice as a priority queue, somewhat contradicting Jones's results. We observed that the high memory overhead of the pointer-based, self-balancing queues translated into poor memory system and overall performance.

And in 2010, Hendriks claimed "[f]or current image analysis programs, the best implementation of that priority queue is the implicit heap. It has the smallest possible memory usage and is faster than all other implementations tested... [except] for very large queue sizes [82]." These conclusions indicate that implicit heaps are appealing for at least some applications. This thesis uses implicit heaps as a software priority queue because they are simple and work well in common cases.

4.1.2 Hardware priority queues

Hardware priority queues motivate the HWDS approach: ENQUEUE and DEQUEUE happen in constant time, whereas the fastest software implementations take logarithmic time for at least one of the two operations. An example hardware priority queue, the shift register priority queue, is shown in Figure 4-1. The shift register priority queue is an array of priority and data payload tuples that are sorted by priority value. A shift register block encapsulates each tuple, and each block connects to its two neighbors. Global lines connect all the blocks to the input and control. Global broadcast lines limit the scalability of the shift register priority queue, but each block makes a decision locally so that sorting happens in parallel. ENQUEUE broadcasts a new tuple to all blocks. Each block sends its current tuple to the left and compares its current priority value, new priority, and priority from the right. If the new priority is less than the current priority, then the block keeps its current data. If the new priority is between the current priority and the priority from the right, then the block latches the tuple. Otherwise, the block latches the right neighbor's tuple. DEQUEUE is simple, with each block sending its tuple to the right and latching from the left. Other hardware priority queue implementations eliminate the global lines—see the discussion in Section 2.1.

EXTRACT can be implemented in the shift-register priority queue by broadcasting both the target payload and priority with a new control signal, and by adding comparators to check the target payload against the stored payload. The target node shifts in its left neighbor. By comparing the priority value, the lower priority nodes will know to shift their values to the right and latch values from the left. Two problems present themselves: the high cost of comparators and insufficient knowledge at nodes that have the same priority value as the target. A solution to the former is to replace the payload with a tag, which can be sized according to the length of the hardware priority queue instead of the size of a

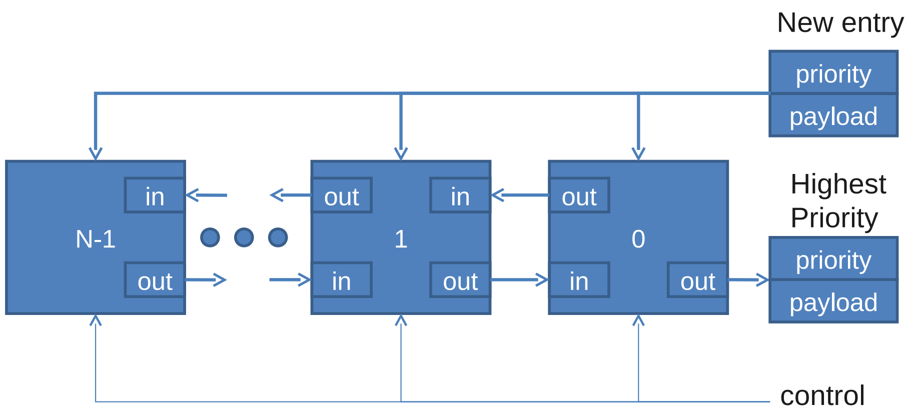


Figure 4-1: A priority queue implemented in hardware.

pointer. This solution increases latency since PEEK needs to translate a tag to a pointer and vice versa for EXTRACT: a CAM can implement tag translation efficiently. Using tags does provide an advantage by reducing the storage and comparison cost for payloads. Sorting nodes that tie in priority by payload (tag) solves the latter, and can be done in parallel with the priority comparisons, so although sorting ties by payload adds work to ENQUEUE, it does not affect latency. However, sorting by payload dictates policy to the priority queue mechanism, which is not in the spirit of this thesis. (As is, the hardware priority queue implements FIFO on ties, which dictates a policy that supports a stable priority queue.) For now, EXTRACT is modeled with the same latency as ENQUEUE.

A systolic priority queue [77] might provide more flexible policies by instructing the nodes lower than the target to shift explicitly, and tag lookup might be pipelined or proceed in parallel with the first systolic block. Future work can evaluate implementations of EXTRACT for different hardware priority queue structures.

4.2 Handling Overflow with a Priority Queue HWDS

The hardware mechanism for `FILL` is simply `ENQUEUE`, but `SPILL` requires an operation that can return a value from an arbitrary position within the HWDS—in particular, the ability to `EXTRACT` the last (lowest priority) node in the queue.

An intuitive solution for overflow handling would use a binary heap as an overflow data structure—Chandra and Sinnen [27] use one. But blindly enqueuing sorted data into a binary heap is wasteful. (Indeed, inserting nodes sorted low-to-high maximizes the work done in a min-heap that inserts nodes at a leaf and heapifies up.) By leveraging the knowledge that the data are sorted, overflow handling can make more intelligent decisions.

Consider instead a sorted linked list implementation of the overflow data structure that merge-sorts overflow nodes. Suppose the number of overflow nodes is k and the size of the overflow data structure is n. The cost of overflow then is approximately $k*lg(n)$ for a binary heap and $k + n$ for a linked list, so when $k > \frac{n}{lg(n)-1}$ the linked list will outperform the binary heap. With an exception-based approach, the amount of work done during overflow (k) should be tuned to amortize the cost of the exception handler while minimizing the future costs of exceptions. With a HWDS of 128 nodes and $k = 64$ so that half the nodes are removed during overflow, the linked list approach should outperform the binary heap for priority queues less than about 512 nodes. In practice the operation costs for the two differ enough that the linked list approach is superior to the binary heap for even larger sizes, but eventually the linear scaling factor of the linked list does limit performance as the size of the priority queue grows.

I implemented both binary heap and linked list overflow data structures. The binary heap implementation is a split HWDS: an overflow data structure that does not take advantage of the HWDS. The linked list implementation is a united HWDS: an overflow data structure that leverages structural locality and the HWDS capabilities.

For a priority queue, the overflow handling needs to be augmented slightly to ensure that ordering violations do not exist between high-priority nodes in the overflow data structure and lower-priority nodes in the HWDS. Hardware modifications are necessary to mark the lowest priority node remaining in the hardware priority queue after spilling. Hardware will also mark nodes when they are enqueued with a lower priority than a marked node. In a shift-register priority queue, this marking requires a node to consult with its right neighbor when latching a new entry. When the head of the priority queue is marked, control logic triggers an underflow exception. The underflow handler fills the HWDS and clears the mark on nodes with higher priority than the lowest priority node remaining in the spill region.

4.3 Experiments

Priority queues are the critical data structure in applications and systems software—some uses include planning, image processing, simulations, timer management, and task scheduling. This section describes two application domains, discrete event simulation and planning, and the experiments conducted to validate and evaluate the contributions of this thesis for handling overflow and sharing for priority queue HWDSs.

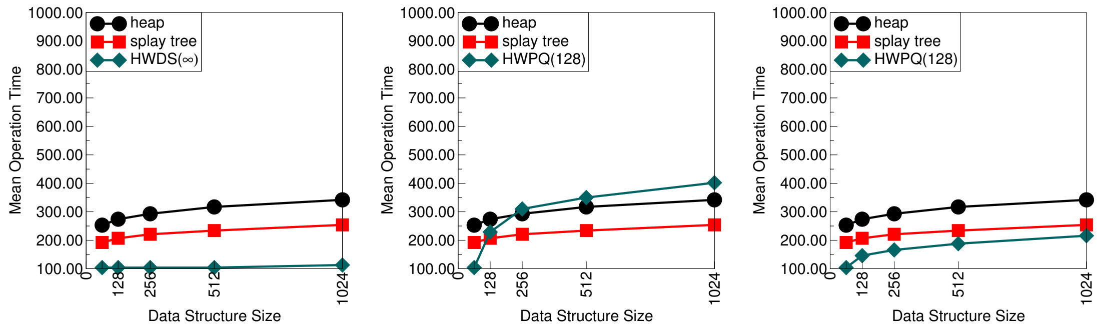
I implemented software-only priority queues and the priority queue HWDS using the experimental infrastructure described in section 3.3. For the software-only priority queue implementations, I implemented a heap (implicit) and a splay tree. I also implemented these priority queues as overflow data structures for split HWDSs, in addition to the linked list united HWDS that is described in section 4.2.

4.3.1 Discrete event simulation

Discrete event simulations can spend up to 40% of execution time managing the pending event set, which is implemented efficiently as a priority queue [105]. Synthetic benchmarks that model the pending event set are used to evaluate priority queue implementations [61, 105].

One model of the pending event set, the classic hold model [61], is useful for benchmarking priority queue performance with a HWDS. A benchmark in the classic hold model executes in two phases: the first phase slowly builds a priority queue to a predetermined maximum size, and the second phase executes a series of *hold* operations—a `DEQUEUE` of the highest priority node, incrementing the priority of the dequeued node, and an `ENQUEUE` of the node. The classic hold model is appropriate for evaluating HWDSs because the maximum size of the priority queue, which is a critical performance parameter, remains fixed throughout the second phase of the benchmark. The variables that affect performance in the hold model of a software-implemented priority queue are its implementation, size, shape (balance), distribution of priorities, and the distribution of priority increment values. A hardware priority queue must consider the maximum capacity of the hardware and the costs for overflow.

In order to evaluate the efficacy of overflow handling and sharing, I implemented a microbenchmark based on the classic hold model and conducted experiments to evaluate the performance of the overflow handling and sharing support described in Sections 3.1, 3.2, and 4.2. I obtain cycle-accurate measurements of execution time that permit calculating a precise average execution time for each insert and hold operation during phase one, and for each hold operation in phase two; smaller numbers are better for the hold model benchmarks.



(a) Infinite hardware. (b) Binary heap overflow. (c) Linked list overflow.

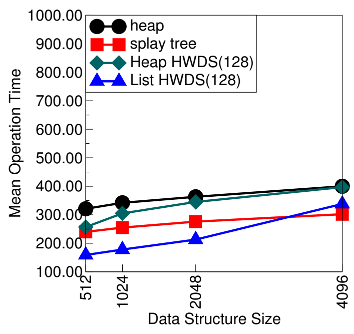
Figure 4-2: Overflow data structure implementation matters. The average hold time of a priority queue HWDS with infinite capacity, a binary heap split HWDS, and a linked list united HWDS compared to software priority queues. Mean hold time is averaged across 1024 hold operations; data structure size is in number of nodes.

Overflow handling for large priority queues

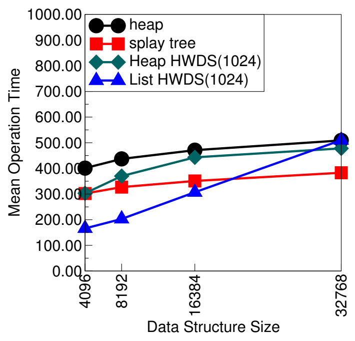
The first set of hold model experiments establish the need for intelligent management of overflow data. These experiments build up the priority queue to a maximum size that varies between 64 and 1024 nodes by powers of 2, and execute n hold operations (where n is 1024 or 16384).

Figure 4-2a shows the obvious benefit when a single application uses an infinite-size HWDS—hardware is faster than software, an unsurprising result. (Note that the average cost is around 100 cycles for infinite hardware because the hold time includes 3 HWDS operations and one arithmetic operation, as well as memory operations to fetch the priority increment amount and benchmark code.) Figure 4-2b is more interesting—it shows how overflow handling using the intuitive approach of a heap as an overflow data structure performs poorly.

If the knowledge that the HWDS contains sorted data is leveraged, a 128-node HWDS outperforms software-only solutions even in the presence of overflow, as shown in Figure 4-2c. If one considered only the intuitive approach, opportunity for improvement from



(a) 128-node HWDS.



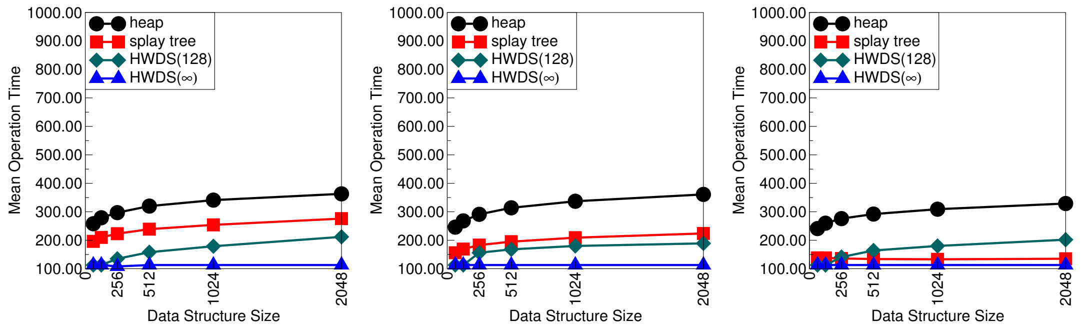
(b) 1024-node HWDS.

Figure 4-3: Performance of software and hardware priority queues averaged across 4,096 hold operations with the same ratio of HWDS capacity to data structure size for 128- and 1024-node HWDSs.

intelligent management of overflow data would be missed. These results indicate that the OS support for overflow handling, in particular the use of a united HWDS, is a useful contribution for improving the performance of at least some kinds of applications that use priority queues.

Figure 4-3 shows how increasing the capacity of the HWDS affects performance, and how the performance trends are similar for a fixed ratio of HWDS capacity to the number of nodes in the data structure. Note that for this benchmark the merge-sorted linked list united HWDS outperforms software when the data structure size is less than 16 times the HWDS capacity for both the 128- and 1024-node priority queue HWDSs. These results demonstrate that larger data structures can be handled by proportionally larger HWDSs using the same policies and OS support as the smaller HWDSs.

The last experiment with a single task accessing an unshared HWDS evaluates the effectiveness of the united HWDS under an increased number of hold operations (16,384) and varying the probability distribution of the priority increment, which is an important parameter for determining performance of a priority queue implementation. Figure 4-4 shows the results for this experiment. With respect to the results shown in Figure 4-2b,



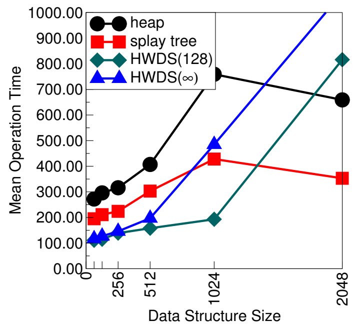
(a) Exponentially distributed priority increment. (b) Biased priority increment (toward FIFO) (c) Bimodal priority increment

Figure 4-4: Performance of software and hardware priority queues averaged across 16384 hold operations.

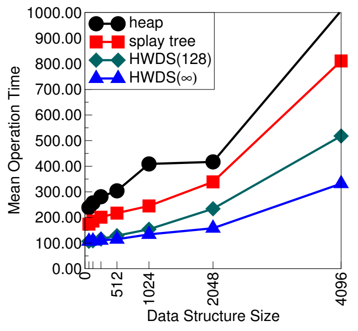
the benefits of the united HWDS persist, or even improve, with more work (hold operations). In terms of the priority increment distribution, the united HWDS does well with an exponential (negative log) distribution and one that is biased toward FIFO behavior—the good performance on the biased distribution may seem surprising, since the biased values ought to cause overflow regularly, but the implementation of the overflow data structure plays a part. The merge sort iterates from the end of the overflow linked list toward the start, and the overflow nodes presumably will be toward the rear of the overflow list because of the bias, so the overflow handler does not need to traverse as much of the data structure. The HWDS outperforms the binary heap with all three distributions, and underperforms the splay tree only with the bimodal distribution.

HWDS assignment for multiple priority queues

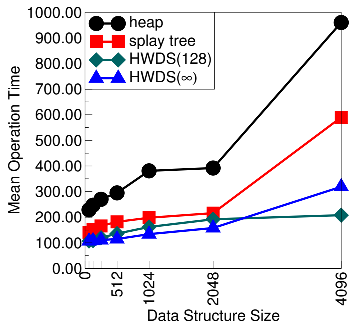
I created two kinds of multi-tasking pending event set benchmarks. The first kind uses tasks that each access a private priority queue of the same fixed maximum size. The second kind also uses tasks that access their own priority queue, but the maximum size varies—in particular, each task has a maximum size exactly half that of the next largest, with a



(a) Same size priority queues, exponential distribution.



(b) Different size priority queues, exponential distribution.



(c) Different size priority queues, priority increment biased toward FIFO.

Figure 4-5: Four tasks sharing a hardware priority queue with 1024 hold operations.

smallest maximum size of 16. Varying the maximum size changes which data structures will benefit most from using the HWDS. The task scheduler is a preemptive time-slicing round-robin scheduler that allocates a 10 millisecond time slice to each task in each round.

Figure 4-5 shows the effect of sharing HWDS resources on both kinds of multi-tasking benchmarks with an assignment algorithm that permits any data structure to utilize the full capacity of the HWDS. Figures 4-5a and 4-5b are the first and second kind of benchmark described in the previous paragraph. Figure 4-5c is the second kind of benchmark, but with a priority increment distribution that is biased toward FIFO queue access. These results show that sharing imposes a cost even for an infinite-capacity HWDS, because the context switch must save and restore data in the HWDS.

In Figure 4-5a, a large spike in performance is seen near the 1024-node priority queue. This spike is due to the increasing cost of context switching, which is causing more context switches to occur because the workload is not finishing as quickly. The performance of HWDS in Figure 4-5b at points 2048 and 4096 owes its performance to the smaller sizes included—the 2048 point includes a 256, 512, and 1024 queue in addition to the 2048, and the smaller queues perform better with a HWDS than with software. The performance of

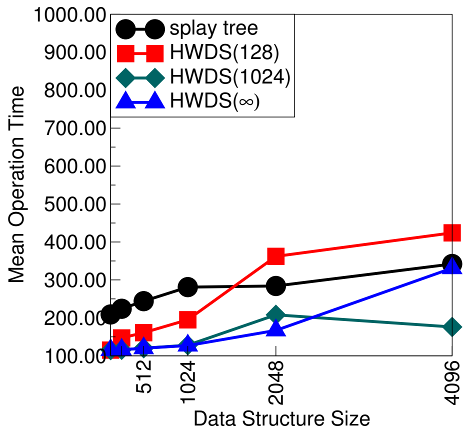
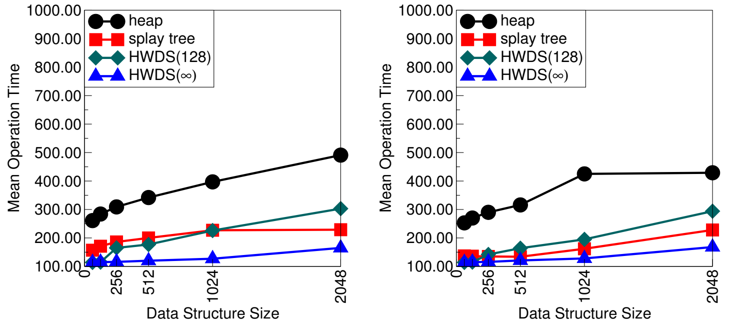


Figure 4-6: Multitask sharing of same-sized priority queues with 4096 hold operations and three sizes of HWDS.

the HWDS when the priority increment is biased illuminates the fact that the size-limited HWDS actually outperforms the infinite-capacity HWDS. The performance benefit is due to the lesser cost of context switching a size-limited HWDS, which motivates experiments in Section 5.3 that limit the permissible HWDS size which a data structure may use.

Figure 4-6 shows how increasing the number of hold operations affects performance for the first kind of benchmark; these results also show how increasing the size of the HWDS shifts the performance curve. Comparing Figure 4-6 with Figure 4-5a, as the number of operations increases, the performance of the HWDS improves (at least for the observed parameter range). The performance of the infinite-capacity HWDS does worse than the 1024-node HWDS for the larger queue sizes because the cost to context switch all of the nodes from the infinite-capacity HWDS is larger than the cost to context switch the smaller HWDS. This performance loss due to context switching justifies HWDS assignment that limits the usable size of a HWDS in order to constrain the context switch cost, which I explore in Section 6.3.

Figures 4-7a and 4-7b show that the biased and bimodal distributions do affect the



(a) Biased priority increment. (b) Bimodal priority increment.

Figure 4-7: Multitask sharing of same-sized priority queues with 4096 hold operations and varying priority increment distributions.

HWDS overflow performance with sharing. Even with the cost of context switching, the HWDS still performs better than a binary heap at the given data structure sizes, although the splay tree does better in general and especially for the bimodal distribution.

The experimental results using multiple tasks that share a HWDS indicate that even simple HWDS assignment can improve performance. The results also motivate further investigation into smarter HWDS assignment, which is described in the subsequent chapters of this thesis.

4.3.2 Planning algorithms

An important algorithm that makes heavy use a priority queue is Dijkstra's shortest-path algorithm, which is used for routing in network devices, navigation in GPS devices, and as a basis for the A* family of path-planning algorithms. Dijkstra's algorithm benefits from CHANGE-KEY, which makes handling overflow more challenging. Chandra and Sinnen [27] show that, if CHANGE-KEY is restricted to increasing priority, then inserting a new node with the updated priority value and allowing the old node to be stale emulates CHANGE-KEY in a hardware priority queue. This solution, however, increases pressure on the hardware priority queue size by adding new nodes when updating a node. Instead, I implemented CHANGE-KEY as a meta-operation that combines EXTRACT followed by ENQUEUE with the new priority; a single macro implements this meta-operation so that the user is unaware of implementation details. Software can implement CHANGE-KEY directly since the meta-operation may be less efficient than a direct implementation for some data structures, for example the binary heap.

To evaluate the cost of overflow handling, I use a version of Dijkstra's algorithm that is executed on real-world maps taken from the 9th DIMACS shortest path implementation challenge benchmarks [26]. For the software data structure implementation, the SmartQ implementation provided with the challenge benchmarks is used. I compare SmartQ with a modified benchmark that uses a hardware priority queue.

The DIMACS GPS benchmarks evaluate both the potential limit of improvement for HWDSs and the performance that is obtained using the overflow support described in Section 4.2 when the capacity of the hardware priority queue is less than the application's data needs.

To find the potential limit of improvement, I instrumented the benchmarks with performance counters to measure the maximum size of the priority queue, the number of priority queue operations (enqueue and dequeue) that execute, and the percent of time that each benchmark spends on priority queue operations. The benchmarks are executed using timing mechanisms that are provided with the challenge code. These timers query the host system for the user time of the process running the application. The timing elides all startup and shutdown costs. To time individual operations, timer calls are added before and after each priority queue operation and ran the application both unmodified and with the timer calls. The difference in total time taken between the two runs is the overhead for making the extra timer calls, half of which is deducted from the sum of the time taken

Table 4-1: Priority queue behavior in selected DIMACS GPS benchmarks

Input	Max Size	Operations	Time
New York City (NY)	925	528693	28.5%
San Francisco (BAY)	886	642540	27.1%
Colorado (COL)	945	871332	30.1%

for priority queue operations (because the time accounted toward the priority queue operations includes half of the timer overhead). Then the ratio of the time taken for priority queue operations to the total time taken by the unmodified application is a measure for the amount of time spent by the application in the PQ.

I gathered performance counters for all of the USA road distance benchmarks in the challenge: Table 4-1 summarizes the measurements for the challenge benchmarks used in the following experiment to evaluate overflow handling for real-world data sets. The full set of measurements is presented in Section 6.4.

The performance measurements show that up to 30% of the execution time of the benchmarks is spent executing priority queue operations. This value gives an estimate of the upper bound of performance improvement from HWDSs.

I also executed modified versions of the smallest three benchmarks with the Simics/GEMS experimental infrastructure. The duration of the benchmarks is reduced to issue 5 path queries; this reduction is necessary so the benchmarks terminate in a reasonable amount of time when executed under cycle-accurate simulation.

Figure 4-8 shows the results from executing these benchmarks, with the performance calculated as a percent improvement versus the SmartQ implementation. Note that the maximum size of the priority queue for these three inputs is less than 1024 (but greater than 800). When the priority queue HWDS is size 1024 there is no overflow and, as expected, the performance improvement is close (within about 4 to 7 percentage points) to the total amount of time spent in the priority queue as measured and reported in table 4-1. More interesting is the performance when overflow does occur. With a 512-node hardware priority queue, the performance of two of the benchmarks is still close to that of the non-overflow. Even when a 256-node hardware priority queue is used, the BAY and COL benchmarks still obtain practical performance improvements. The NY benchmark has negative performance with a 256-node hardware priority queue size that might be attributed to the ratio of priority queue operations to maximum priority queue size.

The experiments with the classic hold model suggested that increasing the number of operations while maintaining the queue size leads to improved HWDS performance, and the same appears to be the case with the GPS benchmark. Finding the ideal ratio would be an interesting study. These results demonstrate that a priority queue HWDS can benefit real-world application software because of the OS support for overflow handling introduced in this thesis.

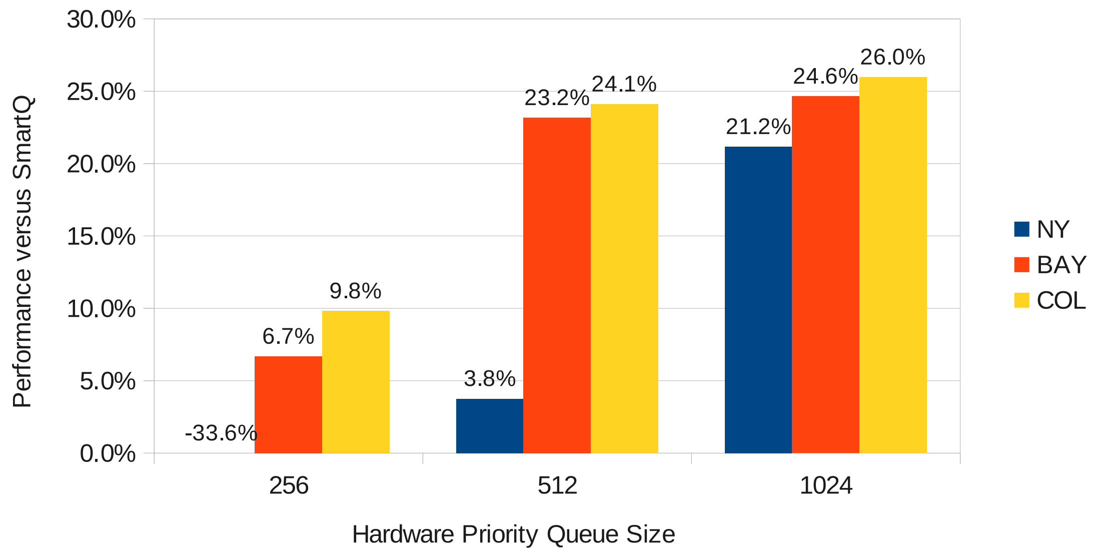


Figure 4-8: Performance of priority queue HWDS as percent improvement over SmartQ with modified—shortened to 5 queries—DIMACS GPS benchmarks.

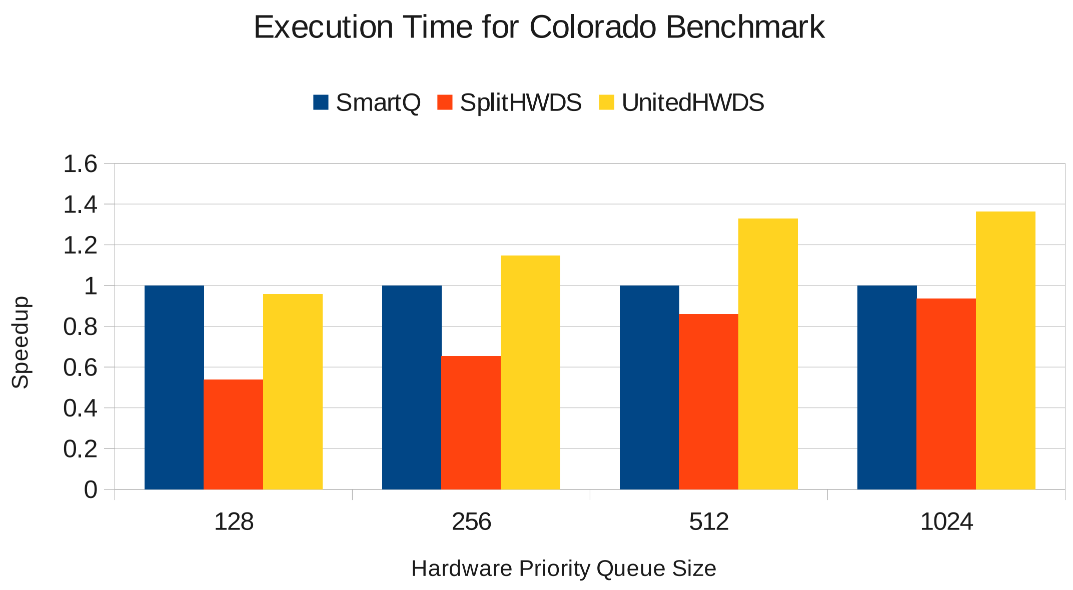


Figure 4-9: Comparison of United HWDS with Split HWDS. Execution time of one iteration of GPS challenge benchmark on Colorado input using the OS support proposed by this thesis (UnitedHWDS) compared to prior art (SplitHWDS) [27] normalized to software-only (SmartQ). Larger is better.

A last experiment with the DIMACS GPS benchmark evaluates how the exception-based united HWDS approach proposed and implemented in this thesis compares with the interposition-based split HWDS proposed and implemented by Chandra and Sinnen [27]. I implemented an interposition-based split HWDS that uses a binary heap as the overflow data structure, and I modified the DIMACS challenge code to use this HWDS and to ignore updates (CHANGE-KEY) to nodes. This implementation is equivalent to what has been proposed in the related work. Figure 4-9 shows the normalized (to the SmartQ) execution times for one iteration of the Colorado GPS challenge benchmark using the SmartQ, the interposition-based split HWDS proposed by others, and the exception-based united HWDS that this thesis espouses; larger numbers are better. For sizes over 128, the united HWDS improves performance as shown earlier. With a HWDS size of 128, neither HWDS approach does as well as SmartQ—indeed, the split HWDS *never does better than software.*

4.4 Summary

This chapter demonstrated the OS support for HWDSs using a well-known HWDS, the hardware priority queue. An EXTRACT operation is proposed for the shift-register hardware priority queue. A united HWDS is described and evaluated, and its performance is compelling on both discrete event simulation and GPS navigation benchmarks using real-world data. HWDS sharing is evaluated with a multitasking benchmark that re-uses the discrete event simulation benchmark framework. The next chapter introduces the map HWDS, which demonstrates how the OS support for the priority queue HWDS translates to another useful data structure.

Chapter 5 – Map HWDS

A map is a data structure that organizes data to support efficient searching. Searching is a fundamental problem in computing: return the node with a specific key from a set of (key, value) nodes. The specified key is the *argument* to the search [68]. Usual operations on a map are:

- `INSERT`: adds a new node
- `EXTRACT`: removes a given node
- `CHANGE-VALUE`: modifies a node's value
- `SEARCH`: finds a node with the given argument

A search can be *exact* or *approximate* if the returned node has the same or closest key as the argument respectively. Keys can have arbitrary length and meaning; common keys include numbers, strings, indices, and hash values. If the search compares key and argument directly then it is a *comparison search*; a *digital search* relies on the binary representation of the argument to find the key. The *skewness* of a search is a measure of the asymmetry of the probability distribution of arguments; text search tends to be strongly skewed, so skewness is an important parameter to consider when evaluating solutions for searching. This thesis considers exact comparison search with numerical keys with varying skewness and maps that use `INSERT`, `EXTRACT`, and `SEARCH`; future work may consider other kinds of search problems and maps that support a `CHANGE-VALUE` operation.

5.1 Software-based Search

Common data structures that support efficient searching are the BST, balanced trees, self-adjusting trees, hash tables, and multiway trees; Knuth [68] describes these in great detail in his textbook. Balanced trees, such as the AVL and red-black trees, ensure $O(\log(n))$ search (and insert, remove) operations. Self-adjusting trees, such as the splay tree, relocate nodes within the tree so that frequently accessed nodes are located nearer the root to improve performance for skew search. Probabilistic search structures, such as the *skip list* [102], use randomization for faster creation and maintenance and provide probabilistic algorithmic performance.

Bell and Gupta [13] evaluated numerical comparison search using BSTs, AVL trees, and splay trees; I adopt their evaluation benchmarks to evaluate the OS support for map HWDSs. Their findings indicate that AVL trees outperform the other trees, although the gap closes when data are skewed. While surprising, their results have also been shown by others for string search: Williams et al. [131] found that BSTs outperform treaps, splay trees, and red-black trees; a modified splay tree does improve over BSTs.

5.2 Map HWDS

Hardware can search small sets of records with numerical keys efficiently with a CAM, but it, like a hardware priority queue, does not support overflow handling or sharing directly. A CAM also does not support direct comparison searches except for specialized uses in which the values are integers that fall within the address range of the CAM; such is the case for the page table-TLB that is described in section 2.1.6. However the solutions presented earlier in chapter 3 do translate to CAM-based (and other) map HWDSs.

5.2.1 CAM-based map HWDS

An implementation of a map HWDS can use a CAM and a fast random-access memory (RAM)—such as SPM—that are the same size. To implement INSERT, the HWDS stores the key in the CAM at an available location, and stores the value in the same location in the RAM, marking the location unavailable. (The entire addressable range is marked available during initialization.) An EXTRACT does a SEARCH for the key, marks the memory at the returned location as available, and returns the node. During a SEARCH, the HWDS control logic passes the argument to the CAM to obtain the location, indexes the RAM at that location to get the value, and returns the node comprising key and value.

5.2.2 Overflow handling

The hardware mechanisms for SPILL and FILL are EXTRACT and INSERT. Unlike the priority queue HWDS, I am unaware of any united HWDS for maps. Therefore any efficient map data structure implements an appropriate overflow data structure. I implemented three such structures: a red-black tree, a splay tree, and a skip list.

5.2.3 Least recently used (LRU) spilling and fill-after-search

Skewed search provides an opportunity for more intelligent overflow handling. In particular, a strongly skewed search will repeat some arguments more often, which indicates that *temporal locality* may be exploited. To evaluate whether temporal locality in overflow handling makes a difference, I re-implemented SPILL to remove the least recently used (LRU) item from the map HWDS and for failover during SEARCH to execute a FILL if the node is found. LRU-based overflow with fill-after search attempts to exploit temporal locality in skewed searches.

5.2.4 Size checks

Even with intelligent overflow handling, when the size of a map exceeds the capacity of the hardware by a sufficient amount the performance of a map HWDS is worse than just using software. I implemented a simple HWDS assignment algorithm that detects if the requested size of a data structure exceeds the HWDS capacity and, if so, assign to software-only. The result of the check hooks software function calls that can either go to a HWDS or a software implementation. Currently this check is done only ahead of time with the cooperation of application software; future work can consider a dynamic change, which would likely demand the use of an interposition-based HWDS or additional hardware support.

5.2.5 Dynamic eviction

When applications SEARCH and EXTRACT in the overflow data structure, the performance of a HWDS suffers, especially with an exception-based HWDS. Another opportunity to improve performance is to detect these conditions and prevent them from happening if possible. A simple, direct method is to evict the data structure from the hardware and rely solely on a software implementation. This method requires an interposition-based HWDS, since otherwise every single data structure operation would cause a failover exception. I implemented a basic interposition-based HWDS to study the effect of dynamic eviction. The HWDS assignment policy using eviction decides to assign a data structure to software-only when an EXTRACT is detected that targets a node in the overflow data structure.

5.3 Experiments

Maps are the critical data structure in applications and systems software—some examples include language interpreters, key-value stores, virtual memory address mapping, schedulers, and timers. This section describes a synthetic search benchmark and experiments conducted to validate and evaluate the OS support for overflow handling and sharing proposed in this thesis for map HWDSs. I implemented software maps and the map HWDS—described earlier in Section 5.2.1—in the experimental infrastructure described in

Section 3.3. The software-only map implementations include a red-black tree, splay tree, and skip list. I also implemented these software maps as overflow data structures for split HWDS.

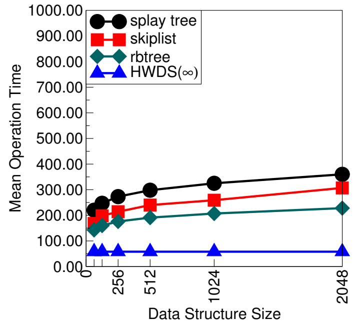
To test the map HWDS, I implemented a synthetic benchmark described by Bell and Gupta [13] that has four steps:

1. Select unique integer keys at random from a uniform distribution.
2. Insert every key in each tree under test and in the *access probability table*, a table containing pairs of key and probability of access that is sorted by probability; probability values affect skewness of key access and are drawn from a modified *Zipf's distribution*.
3. Issue pairs of EXTRACT-INSERT operations and SEARCH operations following an *activity ratio*—the ratio of SEARCH to EXTRACT-INSERT.
4. Record the time consumed during the operations for performance measures.

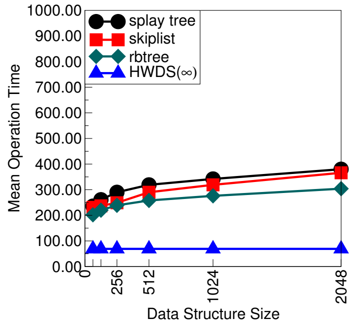
Key, probability selection, and operations are generated offline. Skewness is controlled by the variable α, which yields the uniform distribution when equal to 0 and Zipf's distribution when equal to 1. In experiments using this benchmark, α varies between 0 and 1.420—in general, only the extreme values are interesting, so representative results are shown for α equal to 0, 1.058 (closest to Zipf's), and 1.420.

As with the experiments with the classic hold model described in section 4.3.1, the search benchmark proceeds in two phases of execution. The first phase (step 2 above) builds the map, and the second phase (step 3) modifies and searches within the map. In addition to the established search parameters–activity ratio and skewness—the map's size (number of unique integer keys) is varied to stress the hardware's capacity. Measurements of execution time give the average time for each SEARCH, EXTRACT, and INSERT during the second phase of execution.

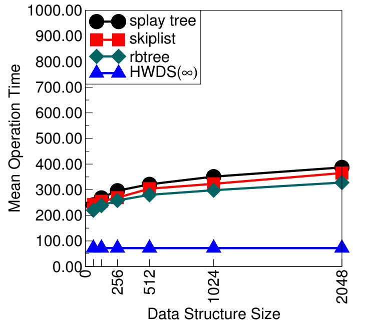
These experiments build up the map to a maximum size that varies between 64 and 2048 by powers of 2, and executes n operations (either 1000 or 4000 for the results presented here) during phase two. An update is counted as one operation, so depending on the activity ratio, the number of total HWDS instructions varies (from $1 * n$ to $1.8 * n$ where n is the number of operations).



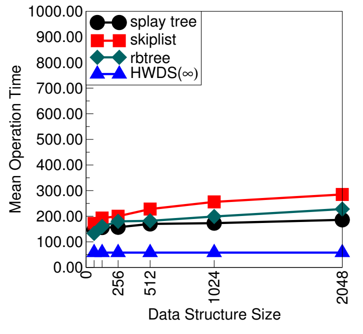
(a) 0% activity ratio, 1000 operations, $\alpha = 0.0$.



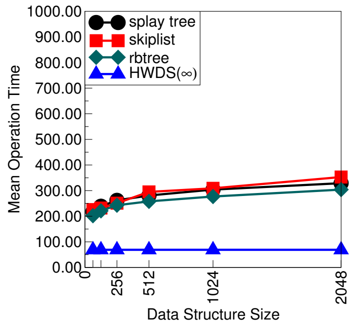
(b) 50% activity ratio, 1500 operations, $\alpha = 0.0$.



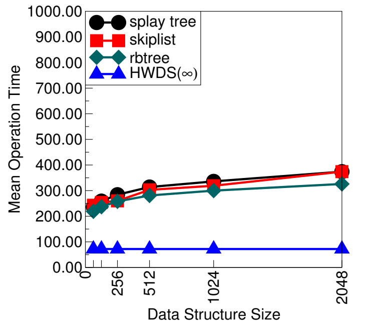
(c) 80% activity ratio, 1800 operations, $\alpha = 0.0$.



(d) 0% activity ratio, 1000 operations, $\alpha = 1.420$



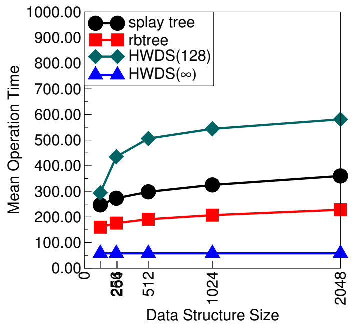
(e) 50% activity ratio, 1500 operations, $\alpha = 1.420$.



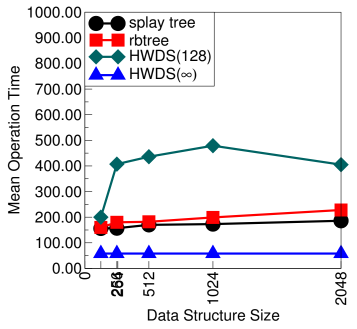
(f) 80% activity ratio, 1800 operations $\alpha = 1.420$.

Figure 5-1: The improvement of infinite hardware and the performance of software map implementations.

The first set of search benchmarks demonstrate the benefits of an infinite-size map HWDS and the relative performance of the three software-only map implementations.



(a) 0% activity ratio, 1000 operations, $\alpha = 0$.



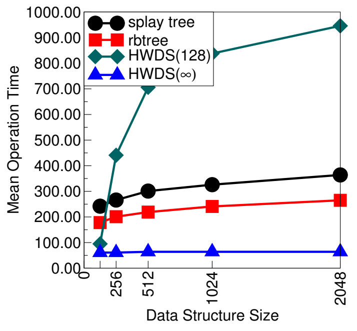
(b) 0% activity ratio, 1000 operations, $\alpha = 1.420$.

Figure 5-2: Overflow handling using the extract-last policy of priority queues with a 128-node map HWDS (HWMAP). (Results with other parameters are similar to Figure 5-2a.)

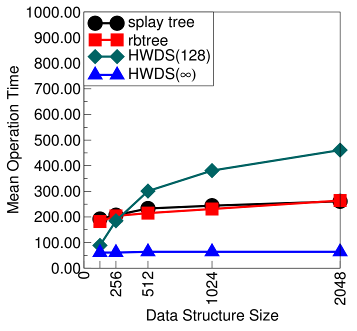
Figure 5-1 shows the benefit when an infinite-size map HWDS is used with and without skewness ($\alpha = 0$ and 1.420). Figures 5-1a, 5-1b, and 5-1c show the results for activity ratios of 0% (no updates), 50%, and 80% respectively with $\alpha = 0$. The average cost with infinite hardware is around 50 cycles because an operation involves one HWDS instruction and the memory accesses to load the code and data for the operation. These charts also show the relative performance of the software-only map implementations, among which the red-black tree performs best in most cases, followed by the skip list then splay tree. On the other extreme, Figures 5-1d, 5-1e, and 5-1e shows how with $\alpha = 1.420$, the performance of the infinite-size HWDS remains the same, but the performance of the software implementations change. These results suggest that the splay tree does better when the map is read-mostly, and the red-black tree does better under heavy updates. The skip list never outperforms the trees and is omitted from the remainder of this thesis.

5.3.1 Overflow handling for large maps

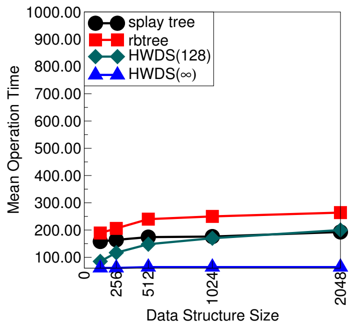
The next set of search benchmark experiments establish the need for intelligent management of overflow. Figure 5-2 shows how overflow handling using the same policy as a priority queue HWDS—spilling the node with the largest key—performs poorly as the map size increases. (Other values of α and activity ratio are similarly bad for the 128-node HWDS. The overflow data structure is a red-black tree.)



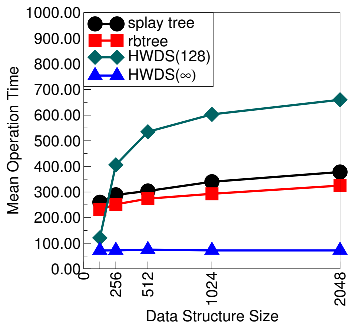
(a) 0% activity ratio, 1000 operations, $\alpha = 0.0$.



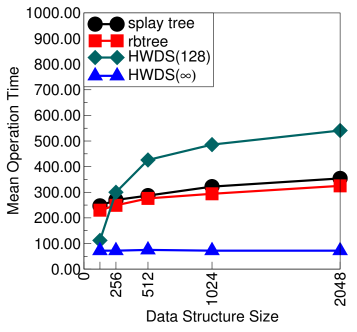
(b) 0% activity ratio, 1000 operations, $\alpha = 1.058$.



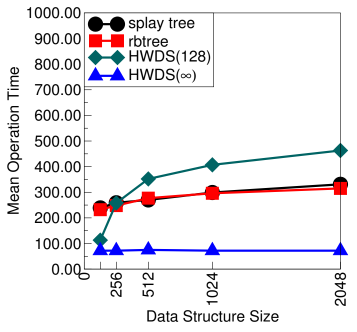
(c) 0% activity ratio, 1000 operations, $\alpha = 1.420$



(d) 50% activity ratio, 1500 operations, $\alpha = 0.0$.



(e) 50% activity ratio, 1500 operations, $\alpha = 1.058$.

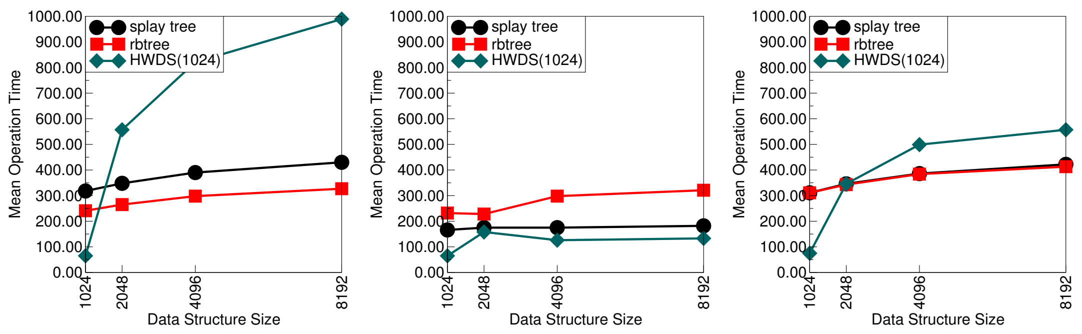


(f) 50% activity ratio, 1500 operations, $\alpha = 1.420$.

Figure 5-3: Map overflow with LRU and fill-after-search, 1000 search operations.

5.3.2 LRU spilling and fill-after-search

Figure 5-3 shows that an LRU-based map HWDS that fills nodes found during a failover search can handle overflow more effectively as arguments become more skewed and activity ratio decreases. This result is not surprising, since a low activity ratio means more searching, for which LRU should be effective, and the more skewness in the search arguments the more temporal locality is available to exploit.



(a) 0% activity ratio, 4000 operations, $\alpha = 0.0$.

(b) 0% activity ratio, 4000 operations, $\alpha = 1.420$.

(c) 80% activity ratio, 7200 operations, $\alpha = 1.420$.

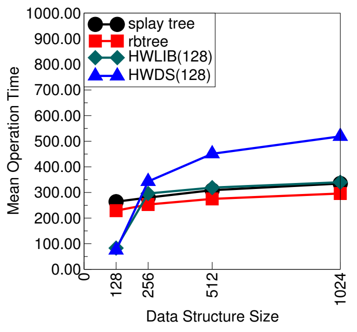
Figure 5-4: Map overflow, 1024-node HWDS, 4000 search operations.

Figure 5-4 shows how a larger HWDS performs with a larger data structure size. Performance is similar between the 128- and 1024-node HWDSs at a given ratio of HWDS capacity to data structure size, except for the read-mostly skewed search which benefits greatly from having a larger HWDS because the increased capacity enables the HWDS to exploit temporal locality better. The map HWDS outperforms software when the ratio of HWDS capacity to data structure size is less than 1.5:1. Note that these results use 4000 search operations rather than 1000; the number of operations had little effect on search performance at this scale, although further experimentation is warranted to determine if the operation count affects performance at larger scales.

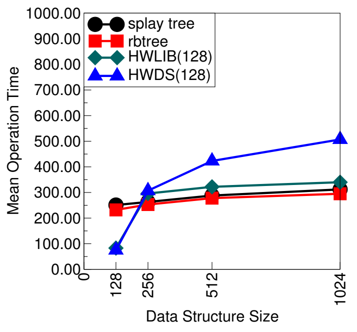
Except for skewed search-only workloads, the map HWDS outperforms software only when the map contains fewer than 50% more nodes than the HWDS capacity. The amount of overflow that can be tolerated is much less than with the priority queue HWDS, and future work should investigate how to increase the amount of overflow that the map HWDS can handle. The priority queue benefits from exploiting structural locality in the united HWDS, and perhaps a similar approach can be developed for the map HWDS.

5.3.3 Eviction

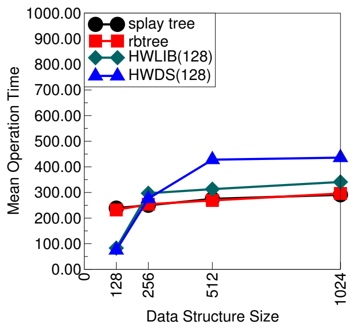
Figure 5-5 shows how an eviction upon EXTRACT HWDS assignment policy can help to curb performance loss when overflow causes failover. With an eviction, the performance of the interposition-based HWDS matches closely with the software-only implementations. These results demonstrate that effective software support can yield performance that achieves approximately the best of both worlds.



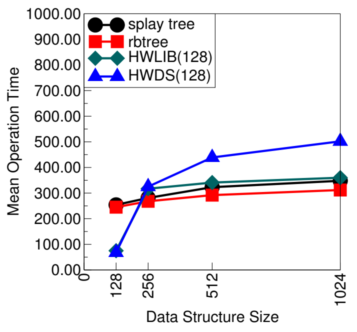
(a) 50% activity ratio, 1500 operations, $\alpha = 0.0$.



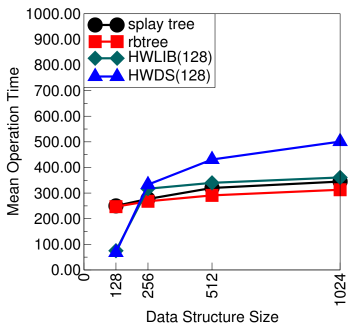
(b) 50% activity ratio, 1500 operations, $\alpha = 1.058$.



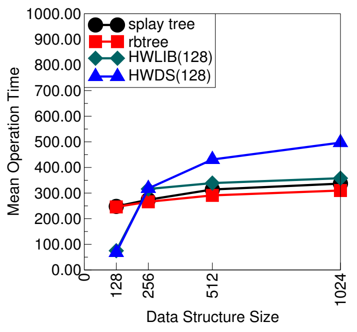
(c) 50% activity ratio, 1500 operations, $\alpha = 1.420$.



(d) 80% activity ratio, 1800 operations, $\alpha = 0.0$.



(e) 80% activity ratio, 1800 operations, $\alpha = 1.058$.

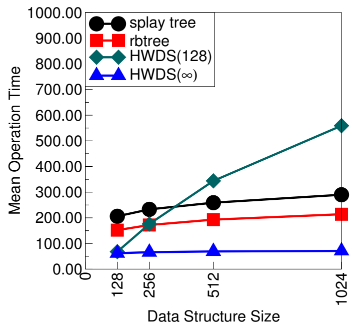


(f) 80% activity ratio, 1800 operations, $\alpha = 1.420$

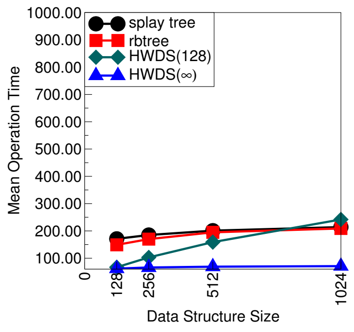
Figure 5-5: Map HWDS overflow handling with HWDS assignment to software upon first EXTRACT.

5.3.4 Sharing for multiple maps

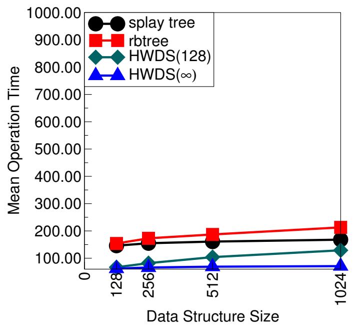
To evaluate sharing map HWDSs, I created separate search benchmarks and placed each within its own task with a task-private map. Each search benchmark has identical parameters except for the maximum map size: each map has a maximum size exactly half that of the next largest, with a smallest maximum size of 16. Varying the maximum size changes which maps benefit from the HWDS. The task scheduler is preemptive time-slicing round-robin with 10 millisecond time slices.



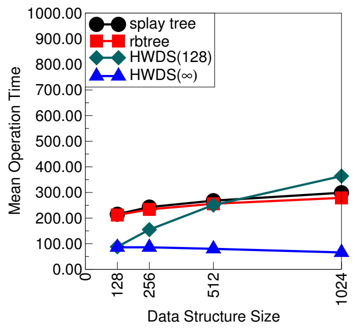
(a) 0% activity ratio, 4000 operations, $\alpha = 0.0$.



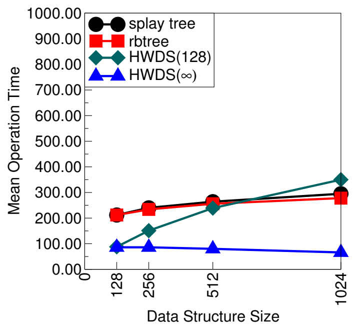
(b) 0% activity ratio, 4000 operations, $\alpha = 1.058$.



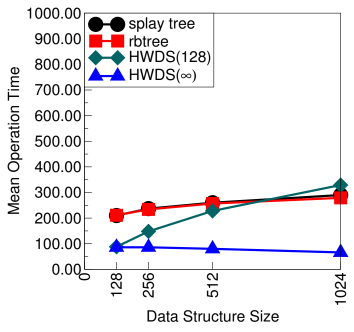
(c) 0% activity ratio, 4000 operations, $\alpha = 1.420$.



(d) 80% activity ratio, 7200 operations, $\alpha = 0.0$.



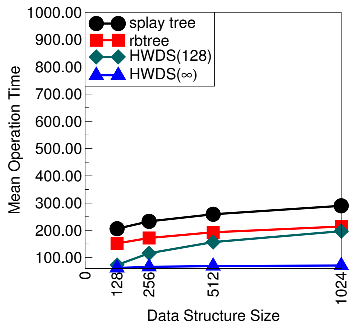
(e) 80% activity ratio, 7200 operations, $\alpha = 1.058$



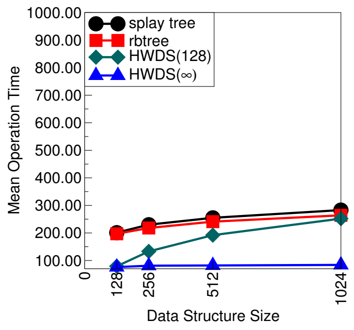
(f) 80% activity ratio, 7200 operations $\alpha = 1.420$.

Figure 5-6: Multitasking search with overflow and different-sized priority queues.

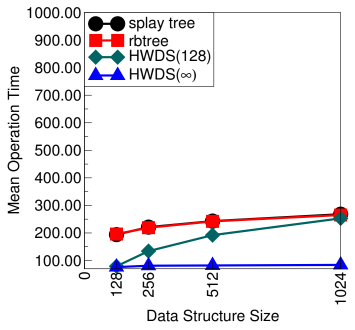
Figure 5-6 shows the performance of an infinite-size map HWDS, software maps (splay tree and red-black tree), and a limited-size, 128-node map HWDS on the multitasking search benchmark. Each power of two over 128 causes another map to overflow; at 256 maximum size, only the one task using a map of that size suffers performance penalties due to overflow. These results show that HWDSs can achieve substantial performance gains versus software, and that as search becomes more skew, the splay tree performance meets that of the red-black tree. Also, when mixing overflow and non-overflow workloads, the HWDS can still perform well in some cases, but eventually does do poorly compared with software-only map implementations.



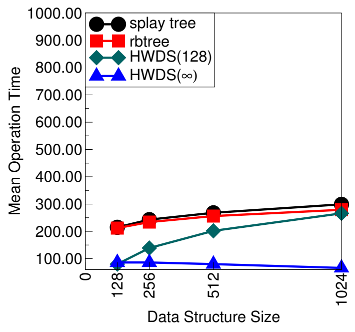
(a) 0% activity ratio, 4000 operations, $\alpha = 0.0$.



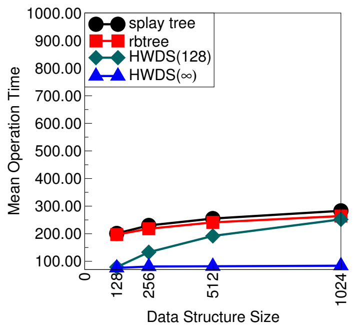
(b) 0% activity ratio, 4000 operations, $\alpha = 1.058$.



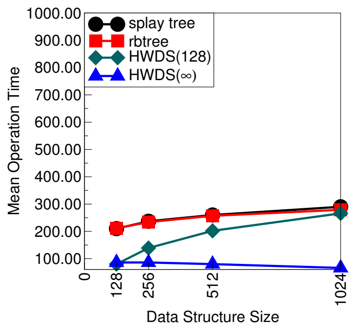
(c) 0% activity ratio, 4000 operations, $\alpha = 1.420$.



(d) 80% activity ratio, 7200 operations, $\alpha = 0.0$.



(e) 80% activity ratio, 7200 operations, $\alpha = 1.058$



(f) 80% activity ratio, 7200 operations $\alpha = 1.420$.

Figure 5-7: Multitasking search benchmarks with size-based assignment and different-sized priority queues.

Figure 5-7 shows the multitasking search benchmarks with a HWDS assignment that uses size checks to avoid using the HWDS in case the requested size of the map data structure exceeds the available capacity of the HWDS. The performance of the HWDS with size-checking assignment is better than simply using software-only. These results show that applications can avoid being wasteful with a HWDS by only using the hardware resources when they will be beneficial. As with the priority queue HWDS, how best to find the best ratio of map HWDS capacity to data structure size is an open question.

5.4 Summary

This chapter presented a map HWDS that uses the same basic style of overflow handling and sharing as the generic HWDS support explicated in Chapter 3. Enhancements were proposed and evaluated, including LRU spilling and filling, dynamic eviction, and size-based HWDS assignment to prevent expensive overflow. Experimental results demonstrate the viability of map HWDSs and the proposed improvements. Some of the remaining issues that are suitable for future work include fleshing out the map HWDS that is sketched in Section 5.2.1, implementing and evaluating the usefulness of the CHANGE-VALUE operation, and inventing a united HWDS for searching.

Chapter 6 – Shared HWDSs for Hard Real-Time Systems

Note: Portions of this chapter were previously published [17].

HWDSs can reduce the latency and jitter of data structure operations, which can benefit real-time systems by reducing WCETs. The OS support for overflow handling and sharing proposed in this thesis permit applications to benefit from HWDSs; this benefit was demonstrated in Chapters 4 and 5. However, real-time applications have different execution requirements than general-purpose applications. This chapter explores those requirements using a priority queue HWDS, presents two novel algorithms for HWDS assignment in a real-time system, and evaluates these algorithms with synthetic task sets and benchmarks modeled from priority queue behavior measured in two applications that are important in real-time and embedded domains: the grey-weighted distance transform for topology mapping and Dijkstra's algorithm for GPS navigation. Experimental results indicate that HWDSs can reduce the WCET of applications even when a HWDS is shared by multiple data structures or when data structure sizes exceed HWDS size constraints.

6.1 Real-time Considerations for HWDSs

Unlike in general-purpose computing, latency, which affects predictability, trumps throughput in a real-time system.

6.1.1 Overflow handling

For real-time systems, the execution time and rate of overflow and underflow exceptions is important because those two parameters affect a task's WCET when using a HWDS. Exception handler execution time depends on the size of the overflow data structure and the number of nodes spilled (equivalently filled). The rate of exceptions depends on two factors: the rate of operations and the number of nodes spilled. The size and rate of operations are application-dependent, but if they are bounded then the exception WCET and rate depends on the amount of work done—the number of nodes spilled.

Tuning the number of nodes spilled by a priority queue HWDS to be any number k less than or equal to half of the HWDS capacity limits the number of exceptions to at most one overflow and one underflow per k operations. In any window of k operations the worst case is that the entire HWDS is full of marked nodes and a PEEK operation is followed by an ENQUEUE. The PEEK induces an underflow exception since the head is marked. The underflow handler fills the HWDS with k nodes and spills at least k nodes, leaving the HWDS in a state with at least k unmarked nodes and possibly marked nodes in the remainder of the HWDS. The HWDS can then satisfy at least k operations without another underflow. The subsequent ENQUEUE may cause an overflow exception which will spill k nodes. At this point the HWDS can satisfy at least k operations without another overflow. Tuning the handlers to spill half of the HWDS size minimizes the number of exceptions taken, which is important because each exception that gets taken adds extra fixed processing overhead to invoke the handler.

Failover exceptions are frustrating for a WCET analysis of HWDSs. In this work on real-time systems, failover is not allowed to happen by tight control of application software. (Additional software engineering is the norm in real-time system development, as is tight hardware-software integration, so the extra control is not an unusual burden to developers.) If the rate of failover exceptions is bounded, they would fit in a WCET analysis similarly to the overflow and underflow exceptions.

6.1.2 Sharing

Sharing is handled similarly to the description in chapter 3, but with two adjustments. First, the HWDS context switch tracks how many nodes it saves, and refills that data structure with the same number of nodes. This adjustment ensures that the same number of nodes are present in the HWDS when the context is restored, an important consideration for bounding the cost of HWDS context switching. Second, a task is only permitted to use one HWDS context; that is, only one of any given task's priority queues may use the priority queue HWDS. This second adjustment aligns the HWDS context switch with the task context switch, which is important when analyzing a task's WCET. The worst case cost of a HWDS context switch is when the HWDS is full and the handler is refilling from a previously full HWDS so that the handler spills and fills the entire HWDS. Similar to overflow handling, the cost of a HWDS context switch depends on the overflow data structure size and implementation, and the number of nodes in the HWDS.

6.2 Response Time Analysis

A HWDS affects task response time by decreasing WCET due to reducing operation latency, but exceptions caused by overflow/underflow conditions increase WCET. Sharing the HWDS among tasks also increases the response time. The following response time analysis evolves a standard response time analysis [11] to include variables that affect WCET when using a HWDS. This analysis only considers periodic tasks.

6.2.1 Notation

- τ: the set of all tasks
- T_i: the i'th task

- p_i: period of T_i
- e_i: the WCET of T_i.
- c_i: the maximum context switch latency of T_i

Usually c_i is equal for all tasks and is included twice in e_i: once for the task preempted by T_i and once for resuming that task.

6.2.2 Standard response time analysis

The response time R_i of T_i is the minimum value of t satisfying

$$t = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k. \tag{6.1}$$

Equation 6.1 considers the WCET of T_i plus the sum of processor time of higher priority tasks overlapping with the time interval t. R_i is found by solving the recurrence

$$t^{(l+1)} = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t^{(l)}}{p_k} \right\rceil e_k$$

starting with $t^{(0)} = e_i$. τ is schedulable if $R_i < p_i$ for all $T_i \in \tau$.

6.2.3 Response time analysis with HWDSs

Adding HWDSs splits the periodic tasks into two sets

- $\widehat{\tau}$: the set of tasks using a HWDS
- $\widetilde{\tau}$: the set of tasks not using a HWDS

so $\tau = \widehat{\tau} \cup \widetilde{\tau}$. HWDS assignment is the problem of choosing whether to place T_i in $\widehat{\tau}$ or in $\widetilde{\tau}$ for every i.

Task response times depend on HWDS assignment. Each task's WCET is now

$$e_i = \begin{cases} \widehat{e_i} + \widehat{x_i} + \widehat{c_i} + \max\limits_{j>i} \widehat{c_j} & \text{if } T_i \in \widehat{\tau} \\ \widetilde{e_i} & \text{otherwise} \end{cases}$$

where

- $\widehat{e_i}$ is the WCET of T_i when the HWDS replaces DS operations
- $\widehat{x_i}$ is the cost of exceptions taken due to using a HWDS
- $\widehat{c_i}$ is the maximum cost to context switch the HWDS for T_i
- $\widetilde{e_i}$ is the WCET of T_i using a software-only DS

$\widehat{x_i}$ depends primarily on how many DS operations can cause exceptions during p_i (i.e. during any job of T_i) and the time needed to handle the exceptions: because $\widehat{x_i}$ depends on the HWDS implementation no generic formula exists for e_i.

$\widehat{e_i}$ depends on $\widehat{c_j}$ for $j > i$, that is the maximum time needed to empty and fill the HWDS of a lower priority task. Preempting a lower priority task j empties j's HWDS and fills i's, whereas resuming j empties i's HWDS and fills j's.

Equation 6.1 still gives R_i but now e_i depends on whether $T_i \in \widehat{\tau}$ or not; that is, on the assignment algorithm. Assignment for just one task depends on whether

$$\widetilde{e_i} > \widehat{e_i} + \widehat{x_i}.$$

Assuming that $\widehat{x_i}$ is bounded then finding the T_i that maximizes

$$\widetilde{e_i} - (\widehat{e_i} + \widehat{x_i})$$

gives the task that will benefit most from using the HWDS.

Including multiple tasks that share the HWDS complicates the assignment problem. In particular $\widehat{c_i}$ varies depending on the cost of emptying and filling the HWDS (i.e. a context switch), so—unlike with traditional response time analysis—a low priority task can affect the response time of higher priority tasks. Conversely higher priority tasks already affect the response time of lower priority tasks. So putting any T_i into $\widehat{\tau}$ necessitates checking whether it negatively affects the rest of the tasks already in $\widehat{\tau}$ in order to find an optimal assignment (see Section 6.3).

6.2.4 Response time analysis with a priority queue HWDS

When using a priority queue HWDS, the costs of $\widehat{x_i}$ and $\widehat{c_i}$ are upper-bounded as follows.

Let $\widehat{S}$ be the size of the HWDS. Tuning the number of nodes that the overflow (underflow) exception handler spills (fills) to be $w < \widehat{S}/2$ guarantees that at most one overflow (underflow) exception will occur for every w priority queue operations (enqueues or dequeues). Let O_i be the maximum number of operations that can occur for any job of T_i, and let $A(w)$ be the WCET of the overflow (underflow) algorithm to handle w nodes. Then

$$\widehat{x_i} < A(w) * \lceil O_i/w \rceil. \tag{6.2}$$

When the context switch invokes the overflow routines to empty the HWDS and the underflow routines to fill it, then the bound on $\widehat{c_i}$ depends on how much of the HWDS T_i uses. Let $\widehat{s_i} <= \widehat{S}$ be the maximum usage of the HWDS by T_i. Then

$$\widehat{c_i} < A(\widehat{s_i}) * \widehat{s_i}. \tag{6.3}$$

For example, if N_i is the maximum size of the priority queue (i.e. maximum number of overflow nodes) then a binary heap implementation of the overflow nodes will have $A(w) \approx$

$w * \log_2 N_i$ (approximating the WCET of the heap by its asymptotic behavior). Then $\widehat{x_i}$ and $\widehat{c_i}$ come directly from Equations 6.2 and 6.3 respectively. In Section 6.4, Software and hardware implementations of priority queues are measured for the WCET of their ENQUEUE and DEQUEUE operations and—for HWDSs—SAVE-CONTEXT, RESTORE-CONTEXT, SPILL, and FILL. I evaluate HWDS assignment algorithms with those measurements.

6.3 HWDS Assignment for Real-time Systems

I use terminology from scheduling to describe HWDS assignment for real-time systems—indeed the assignment problem is similar to the problem of task scheduling. An assignment is *feasible* if a solution to Equation 6.1 can be found for every task (equivalent to finding a feasible schedule). If an assignment algorithm exists that produces a feasible assignment for a set of tasks, then those tasks are *schedulable*. An assignment algorithm is *optimal* if it always produces a feasible assignment for a set of tasks when one exists.

I evaluate four assignment algorithms for HWDSs: software-only assignment (SOA), hardware-only assignment (HOA), priority-aware assignment (PAA), and context switch cost-aware assignment (CSCAA). The first two algorithms are naïve and represent two extremes, and the latter two are greedy algorithms employing different heuristics to make choices about when to use a HWDS. None of these algorithms is optimal, and the PAA and CSCAA algorithms do not permit tasks to change their priorities.

Some aspects of these algorithms are dependent on data structure behavior in particular on the WCET of HWDS operations, exceptions, and context switches. A priority queue HWDS has a bounded WCET if the maximum priority queue size, maximum number of operations per period, and the HWDS size are bounded. In general these algorithms will work for any HWDS that has bounded WCET based on the data structure size and operations. If a HWDS requires more information to bound its WCET, then new algorithms may be required. Future work should evaluate the difficulty of HWDS assignment and whether efficient

The SOA algorithm simply assigns every task to use a software-implemented DS: the SOA algorithm ignores the HWDS.

The HOA algorithm assigns every task to use the largest possible HWDS. Usually the largest available HWDS gives the best performance out of all the available HWDS sizes, but not always. As the usage of the HWDS increases, the rate of exceptions should go down assuming that the work done during the exception handler increases. However the latency of the exception handlers will increase, and so will the HWDS context switch due to needing to move more data. For small numbers of operations per period, the larger HWDSs underperform smaller HWDSs; at small counts of data structure operations software typically performs better than any HWDS.

The PAA algorithm (Algorithm 1) iterates through tasks from the lowest priority to the highest priority choosing at each task whether to use the HWDS by comparing the WCET of the software implementation with the WCET of the HWDS. This algorithm tracks the maximum HWDS context switch of the tasks that it has assigned to the HWDS so that it can compute the WCET accurately taking into account the context switch costs of lower-priority tasks. Iterating from low to high priorities allows the algorithm to move in one direction. The reason that this algorithm is not optimal is that higher-priority tasks that use the HWDS have a WCET that depends on whether (and which) lower-priority tasks use the HWDS. Because the algorithm only moves in one direction, it does not allow for re-evaluating the assignment of lower-priority tasks, and therefore can miss feasible assignments.

CSCAA (Algorithm 2) is similar to PAA except for the cost heuristic that gets added to the HWDS WCET. The cost heuristic penalizes low-priority tasks for using the HWDS.

Algorithm 1: Priority-Aware Assignment (PAA)

```
Input: $n$: number of tasks, $\tau$: task set, $N$: max DS sizes, $O$: max DS operations, $S$: max HWDS size
$\widehat{\tau} = \emptyset$
$\widetilde{\tau} = \emptyset$
$\widehat{c_m} = 0$
for $i$ from $n$ to $0$ do
    $\widehat{e_i}$ = get_hwds_wcet ($N_i$,$O_i$,$S$,$\widehat{c_m}$)
    $\hat{S}_i = S$
    for $s < S$ do
        $\widehat{e_i}$ = get_hwds_wcet ($N_i$,$O_i$,$s$,$\widehat{c_m}$)
        if $e < \widehat{e_i}$ then
            $\widehat{e_i} = e$
            $S_i = s$
        end
    end
    $\widetilde{e_i}$ = get_swds_wcet ($N_i$,$O_i$)
    if $\widehat{e_i} < \widetilde{e_i}$ then
        add_to_set ($\widehat{\tau}$, $T_i$)
        if $\widehat{c_i} > \widehat{c_m}$ then
            $\widehat{c_m} = \widehat{c_i}$
    else
        add_to_set ($\widetilde{\tau}$, $T_i$)
end
return $\widehat{\tau}$,$\widetilde{\tau}$
```

This heuristic tries to offset the effect of lower-priority tasks on higher-priority tasks. In particular, the WCET of high-priority tasks affects low-priority task response times, so reducing high-priority task WCETs should benefit response times for a set of tasks. Of course, the penalty may prevent low-priority tasks from using the HWDS when they could (and should), so this algorithm can miss feasible assignments. The cost heuristic can be any function that gives a penalty to a task that—if it uses the HWDS—would increase the maximum HWDS context switch time compared to tasks with a lower priority. For this work, I used a cost heuristic that multiplies the amount a task will increase the maximum HWDS context switch latency times the number of tasks with a higher priority: In Algorithm 2 the function `get_cost` returns $(c_i - c_m) * (n - i)$ or 0, whichever is greater.

Algorithm 2: Context Switch Cost-Aware Assignment (CSCAA)

```
Input: $n$: number of tasks, $\tau$: task set, $N$: max DS sizes, $O$: max DS operations, $S$: max HWDS size
$\widehat{\tau} = \emptyset$
$\widetilde{\tau} = \emptyset$
$\widehat{c_m} = 0$
for $i$ from $n$ to $0$ do
    $\widehat{e_i}$ = get_hwds_wcet ($N_i$,$O_i$,$S$,$\widehat{c_m}$)
    $\widehat{S_i} = S$
    for $s < S$ do
        $e$ = get_hwds_wcet ($N_i$,$O_i$,$s$,$\widehat{c_m}$)
        if $e < \widehat{e_i}$ then
            $\widehat{e_i} = e$
            $S_i = s$
        end
    end
    $\widetilde{e_i}$ = get_swds_wcet ($N_i$,$O_i$)
    if $\widehat{e_i}$ + get_cost ($i$,$n$,$S_i$,$\widehat{c_m}$) ¡ $\widetilde{e_i}$ then
        add_to_set ($\widehat{\tau}$, $T_i$)
        if $\widehat{c_i} > \widehat{c_m}$ then
            $\widehat{c_m} = \widehat{c_i}$
    else
        add_to_set ($\widetilde{\tau}$, $T_i$)
end
return $\widehat{\tau}$,$\widetilde{\tau}$
```

6.4 Experiments

I conducted a series of experiments to evaluate HWDSs in the context of hard real-time systems. These experiments use a priority queue HWDS, synthetic task sets to explore the parameter space of the HWDS as the parameters relate to WCET, and workloads that approximate real-world applications. Experiments are conducted using the experimental infrastructure described in Section 3.3.

I measured values for WCET parameters that underlie all of the following experiments. To estimate the WCET of priority queue operations I implemented an implicit binary heap as a representative software priority queue. I designed a series of measurement tests that build a priority queue up to a specified size, and then measure the cost of an operation at that size. Five specific events are measured in isolation: enqueue, dequeue, overflow exception, underflow exception, and HWDS context switch. The latter three are only relevant and measured for a HWDS. All caching is disabled to obtain the WCET of these five events. Although these measurements are pessimistic, the lack of a time-predictable cache is problematic. As a result, memory access latency dominates the WCET measurements.

To force the worst-case conditions for the software priority queue, measure an ENQUEUE of a node with priority less than the highest-priority node in the heap so that the ENQUEUE must move the new node to the top of the heap resulting in a maximum number of swaps (equal to the log base-2 of the priority queue size). A DEQUEUE of the minimum value causes a maximum amount of work in a heap.

For the HWDS ENQUEUE and DEQUEUE WCET, the HWDS must be in a state that will not cause an exception. Before measuring ENQUEUE, ensure the HWDS has enough spare capacity to accept the new node, and before measuring DEQUEUE ensure at least one valid node is at the head of the queue. To generate the WCET overflow, the nodes that get spilled must cause the spill algorithm to do maximum work. Using the united HWDS described in Section 4.2, spilling iterates from the tail of the linked list to the head (which has highest priority); to cause the WCET overflow, empty the HWDS and then fill it with new nodes that have priority less than the head of the overflow linked list, thus ensuring that the spill algorithm iterates through the entire linked list before completing.

The underflow handler has a special condition under which it has to spill nodes; when the HWDS is full of marked nodes, it must fill from the spilled nodes and also spill some of its marked nodes. The worst-case condition of an underflow is generated by enqueueing nodes with priority less than the head of the spilled nodes (as with the overflow case), marked all nodes in the HWDS, and then issued a DEQUEUE. The DEQUEUE causes an underflow, and the exception handler finds that no capacity exists to fill, so it spills nodes. The spills will take maximum time because the handler spills nodes with higher priority than the nodes already in the spill data structure. The underflow handler eventually fills the HWDS.

To cause the WCET of the HWDS context switch, fill the HWDS to its maximum size using two separate data structures while ensuring the HWDS contains nodes with priority less than the head of the spilled nodes. Then cause a HWDS context switch by issuing an operation for the priority queue that is not currently in the HWDS context. The context switch handler spills all of the nodes in the HWDS, which (because of the ordering of nodes) takes maximum time, and then fills the HWDS with nodes from the requested priority queue's overflow data structure.

6.4.1 Schedulability

I designed a series of experiments using synthetic task sets to characterize the HWDS parameter space and evaluate the HWDS assignment algorithms A task set is started by creating a set of n tasks choosing integer task periods p_i uniformly from $[1, 1000]$. Choose task utilizations u_i uniformly at random from $[0.001, 1)$ implicitly selecting task execution times e_i. After assigning all n tasks a utilization, normalize each u_i so that $\sum_{i=0}^{n} u_i = U$, where U is some target utilization value. This method of generating tasks provides a variety of task sets while controlling the number of tasks and the task set utilization. Use response time analysis (Equation 6.1) to ensure the generated task set is schedulable, and regenerate any sets that fail the schedulability test. Then modify each generated task set to include priority queue operations parametrized by a max priority queue size, max HWDS size, priority queue implementation, and number of operations to complete in a period. Using the task's period and utilization, calculate compute time and add the WCET determined by the priority queue parameters. Priority queue size and implementation determine the WCET for any operation, and the priority queue size with the number of operations determines the WCET for the HWDS exceptions. The HWDS and priority queue sizes determine the WCET for the HWDS context switch.

The parameters of max priority queue size, priority queue implementation, and number of operations are varied in a controlled way. For each particular assignment of parameters, generate 10000 task sets and attempted to assign priority queue usage for each task set using all four of the algorithms (SOA, HOA, PAA, and CSCAA) presented in Section 6.3. For each task set and assignment algorithm, determine whether the task set is schedulable after priority queue assignment. For these experiments, I set the max HWDS size at 1024 and let PAA and CSCAA choose to limit individual tasks to a smaller size; in practice these algorithms typically—but not always—use the largest possible HWDS size.

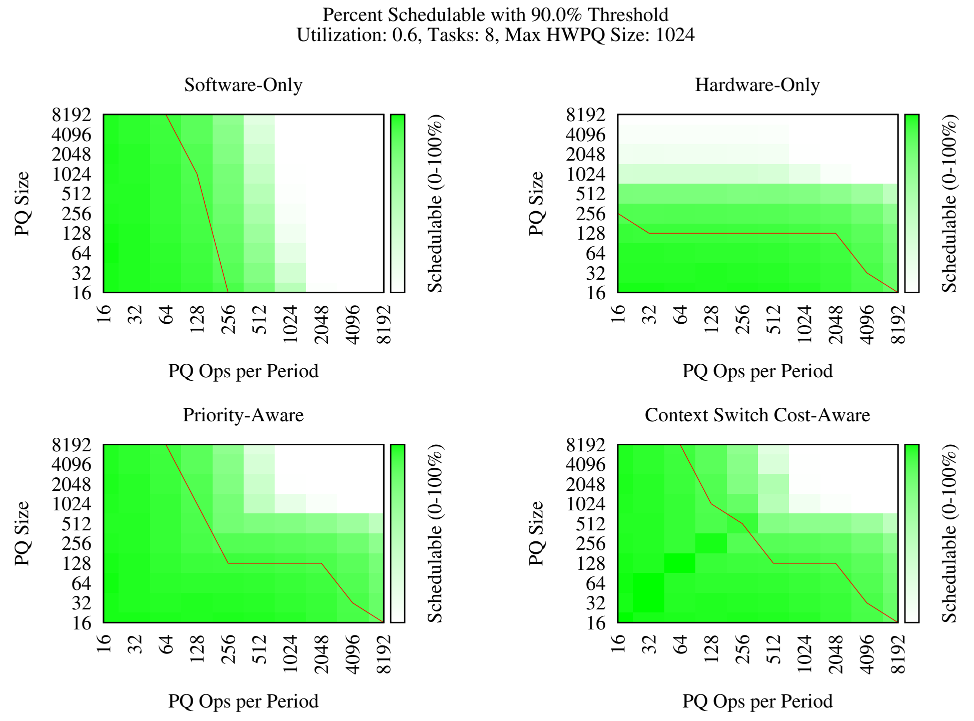
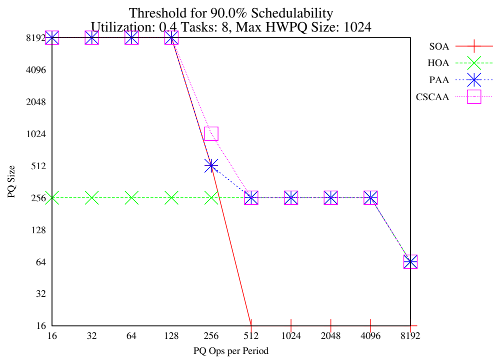
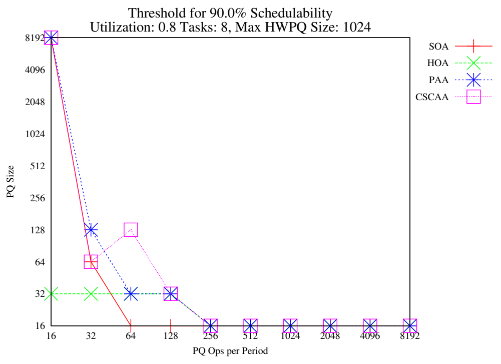


Figure 6-1: Schedulability of random task sets for utilization (without priority queue operations) fixed at 0.6 and task set size at 8. Varying utilization and the number of tasks moves the threshold lines, which are shown in later figures.

Figure 6-1 shows the results as both the max priority queue size and the number of priority queue operations per period vary by powers of 2 from 16 to 8192. For this particular figure, the task set utilization U is 0.6 and the number of tasks per task set to 8. The plot shows the percent of task sets (out of 10000) that are schedulable after assignment for each combination of priority queue size and number of priority queue operations. The threshold line plot delineates an upper limit below which each combination feasibly schedules at least 90% of its task sets. These results show how the different assignment algorithms work, and in particular show that PAA dominates SOA and HOA for much of the explored space. The threshold line also shows that differences exist between the schedulability of task sets assigned using PAA versus CSCAA, with neither outperforming the other for all parameters although CSCAA generally does better than PAA.



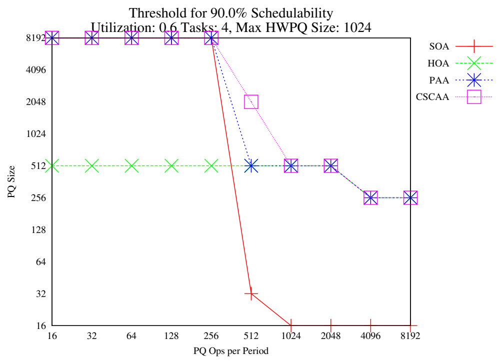
(a) Schedulability with $U = 0.4$.



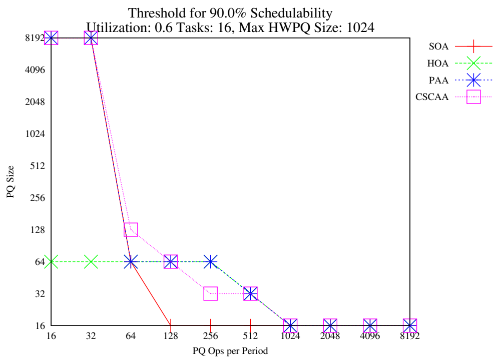
(b) Schedulability with $U = 0.8$.

Figure 6-2: As utilization decreases (increases), threshold lines move up (down) because applications have more (less) spare utilization to accommodate priority queue operations.

Figure 6-2a shows just the threshold lines this time for a task set utilization U at 0.4, and again with the tasks fixed at 8; Figure 6-2b shows how increasing U affects schedulability by measuring schedulability with U at 0.8 and with 8 tasks. When system utilization is low the extra slack available in the system allows for priority queue operations to use more time, which leads to more task sets being schedulable. In general, the threshold lines move up indicating that for a given number of priority queue operations, the task sets having priority queue sizes twice as large are schedulable over 90% of the time with the extra 20% available CPU time.



(a) Schedulability with 4 tasks.



(b) Schedulability with 16 tasks

Figure 6-3: As the number of tasks decreases (increases), the threshold lines move up (down). Halving (Doubling) the number of tasks more than doubles (halves) the number of schedulable task sets.

Figure 6-3a again shows the threshold lines, this time with U at 0.6 and with 4 tasks; Figure 6-3b shows how increasing the number of tasks with fixed U affects schedulability by keeping U at 0.6 and increasing the number of tasks to 16. The extra tasks increase the global number of priority queue operations (since every task does the same workload). Doubling the tasks has the effect of reducing by a factor of two the priority queue sizes of tasks sets that are schedulable at least 90% of the time for a given number of operations (two factors if compared to half as many tasks and 20% more CPU time).

6.4.2 Real-world Applications

The synthetic task sets demonstrate priority queue HWDSs with the PAA and CSCAA algorithms can decrease utilization hence increase schedulability of applications that use priority queues. This section shows how HWDSs might benefit real-world applications, which may not exhibit behavior that is similar to the synthetic task sets. Two important application domains in real-time and embedded systems are navigation and terrain mapping. Both of these domains contain applications that use a priority queue as a central data structure in their main algorithms. From the navigation domain, I use a version of Dijkstra's algorithm that is executed on real-world maps taken from the DIMACS shortest path implementation challenge benchmarks [26]. From the terrain mapping domain, I use an implementation of the grey-weighted distance transform that executes on a random 3D image; this application has been used previously to evaluate a variety of software priority queues [82]. I call these applications GPS and GWDT respectively. Both applications and their inputs are available online, see [82, 26].

In order to simulate these real-world applications, I measured their behavior with respect to PQ parameters that affect HWDS WCET. These measurements are the same as those used in Section 4.3.2, where the methodology for taking measurements is explained for the GPS application. Table 6-1 summarizes the measurements. For the GWDT application, I included the `PEEK`, `ENQUEUE`, and `DEQUEUE` operations with priority queue memory management; the software priority used for the measurements was the 4-heap [82].

Using the parameters measured from running the applications, I modeled two new applications that simultaneously run x numbers of small (32 pixel) GWDT tasks, y numbers of local GPS search tasks, 1 large (64 pixel) image processing task, 1 regional GPS search task, and 1 long-distance GPS search task. One application lets x vary from 0 through 12 with y fixed at 1 (call it the GWDT application), and the other application lets y vary from 0 through 12 with x fixed at 1 (call it the GPS application). The total number of tasks in either application varies from 4 to 16.

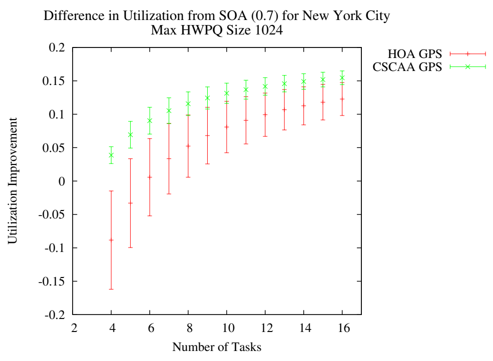
For each application at a given number of tasks, 10000 random task sets are generated

Table 6-1: Priority queue behavior in real-world applications.

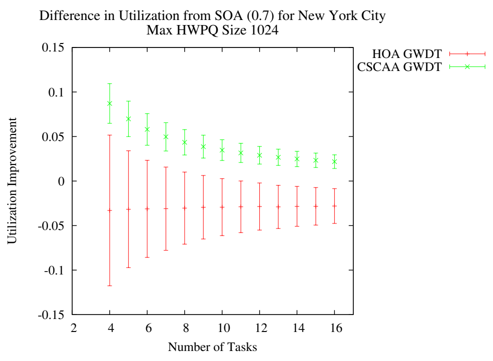
App.	Input	Priority Queue Size	Operations	Time
GWDT	32 pixels	16303	168840	31.4%
	64 pixels	56447	1353326	33.5%
GPS	NYC	925	528693	28.5%
	S.F. BAY	886	642540	27.1%
	Colorado	945	871332	30.1%
	Florida	1413	2140753	28.4%
	NW US	1723	2415891	29.2%
	NE US	1796	3048907	26.7%
	California	2355	3781631	27.4%
	Great Lakes	1810	5516239	27.9%
	Eastern US	2336	7197247	24.6%
	Western US	4281	12524209	24.3%
	Central US	5086	28163632	22.4%

with the utilization drawn randomly as before (uniform in $[0.001, 1]$ then normalized to a target U after all tasks have a utilization), but now with the period determined by the measured priority queue parameters. In particular, the WCET of a software priority queue is determined (using measurements from the implicit binary heap) for the maximum priority queue size and number of priority queue operations for the task, and uses the percent of time the task should spend on the priority queue to determine how long its total compute time should be. Then the task's period is computed by dividing its total compute time by its randomly generated utilization. Any task set that does not pass the response time analysis is regenerated.

The result of task set generation is a set of tasks that use a software priority queue and whose task set has a utilization equal to a known value U. The software priority queue WCETs is then removed from the tasks and run each assignment algorithm (SOA, HOA, PAA, and CSCAA) on the task set. The SOA algorithm will result in a schedulable task set with a utilization equal to U. Instead of using schedulability as the metric for performance in these experiments, the amount the assignment algorithm improves (or degrades) task set utilization is used; an improvement in utilization is a positive number, so larger is better, and negative numbers indicate that the assignment algorithm does worse than SOA.



(a) Utilization improvements for GPS.



(b) Utilization improvements for GWDT.

Figure 6-4: Utilization improvements with increasing numbers of tasks executing local search in NYC or small input for GPS and GWDT applications respectively.

Figure 6-4a shows how HOA and CSCAA improve utilization over SOA for the application that varies the number of tasks running a local GPS search; each point is the arithmetic mean of the difference between the utilization of SOA—fixed at 0.7—and one of the assignment algorithms (either HOA and CSCAA) averaged across 10000 trials, and with error bars showing the sample standard deviation in both directions (one standard deviation up and one down). The local GPS search is executing the benchmark challenge for New York City, with the regional and long-range searches executing the northeastern US and eastern US benchmarks respectively. Figure 6-4b shows the same measurements but taken as the number of tasks running the small (32 pixels) GWDT (32 pixels) increases. The results for PAA are not shown because they overlap closely with those for CSCAA. The gains for the GPS application are around 10–16% utilization which represents an improvement of 14–22% over the software PQ utilization.

The real-world applications demonstrate some interesting results. First is that just using a HWDS (HOA) yields large swings in utilization; the smallest GWDT task has a standard deviation of around 7% utilization. Second is that for some applications the benefit of using HWDS may actually increase as the number of tasks increases; conversely the benefits may decrease, as shown by the GWDT results. Even so, the CSCAA algorithm produces useful HWDS assignments in these real-world task sets and improves task set utilization, which enables real-time developers to schedule more hard real-time tasks. The extra utilization also could be useful for admission control of sporadic and aperiodic tasks.

6.5 Summary

This chapter demonstrated that HWDSs can benefit real-time systems by reducing WCETs even when data structure sizes exceed the size of the HWDS. Systems software support provides flexibility to remove size and sharing limitations of hardware so that applications can benefit from using HWDSs. I devised two new algorithms that assign tasks to use either a HWDS or a software-implemented data structure, and experimental results show those algorithms outperform just using the software or just using the HWDS for much of the explored application and parameter space. A priority queue HWDS shows how real-world applications for navigation and image processing could obtain practical improvements in the range of 5–15% of total utilization using the intelligent approaches to overflow handling and HWDS assignment proposed in this thesis.

Chapter 7 – Future Work and Conclusion

Before I conclude my dissertation thesis, this chapter identifies some of the possible directions for future work.

7.1 Policies for Accessing Memory

Decades of research on caching has explored policies to improve performance: penalty-reducing algorithms like critical-word first, early restart, read prioritization, and write merging; miss-reducing techniques like hit under miss, hardware prefetching, and cache pinning; reducing access latency with virtual addressing, cache sizing, and pipelining; and eviction algorithms like LRU, least frequently used, and victim caching. Parallels to these improvements may exist for HWDSs, since they too provide an interface to memory. The solutions likely differ, because—as demonstrated by the overflow handling for the priority queue united HWDS—overflow data structure implementation may affect policy and algorithm performance.

7.2 HWDS Assignment

This thesis shows that assignment algorithms make a difference. Better algorithms, both static (offline) and dynamic (online), certainly exist for HWDS assignment, and evaluating their complexity and effectiveness is a promising avenue for future research. An important open question related to HWDS assignment is what size HWDS should a data structure of a given size be assigned.

7.3 Data Sharing

Nothing prevents tasks from sharing a HWDS with the same data structure. However such sharing imposes two new requirements on the HWDS: synchronization and protection. The synchronization and protection of shared data are well-studied problems. Synchronization is solved with mechanisms such as locking and TM. Protection is provided by OS support for private address spaces and shareable regions within those spaces; for example, shared pages in a page-based VMA space. This thesis supports only task-private data structures, so synchronization is not a problem, and protection is implicit.

HWDSs may prove beneficial for data sharing, because the hardware could implement its own synchronization primitives. Protection does not seem problematic, because task context can include HWDS context, in which case the OS only needs know which HWDS contexts a task may access. A deeper study of data sharing is needed to test these hypotheses, but the idea seems promising.

7.4 OS Optimizations for HWDSs

HWDS exceptions and hardware performance counters offer new knowledge about data structure usage that the OS might use advantageously. Some optimizations to investigate include deferring exception handlers, lazy HWDS context switching, co-scheduling tasks that share a data structure, avoiding preemption for a minimum time after a HWDS context switch, pinning data structures to the HWDS for high priority tasks, prefetching HWDS context for tasks near the front of the scheduler's ready queue, and using the overflow data structure when it is already in cache. All of these optimizations have potential to improve the performance of multitasking systems using HWDSs.

7.5 Integration with Programming Languages and Libraries

From a programmer's perspective, libraries—like STL—and the OS could use HWDSs independent of applications to replace the use of data structures. Compilers can play a role in effective HWDS use as well. Code generation and optimization for HWDS instructions is an open area of research.

Object-oriented languages support abstractions and libraries for the data structures and operations used throughout this thesis: the C++ STL provides priority queue and map *containers*—a container is the STL equivalent of an abstract data type. STL containers implement data structure operations as member functions of the container template class. Some of these functions already are supported by HWDSs to a limited extent—for example insert, erase, find, begin, push_front, and pop_front. Other functions can be provided with trivial hardware modifications, for example hardware counters can implement functions related to capacity such as size, max_size, and empty. The difficulty in providing the remaining functions is that either the hardware support is non-trivial or the nodes' values must be accessed, which requires knowledge about the object layout. Also unclear is how to handle iterators, which permit applications to retain handles to the container. Is the gap between the HWDS interface and common library interfaces such as the STL bridgeable? Is a light-weight portability interface that can span multiple languages and libraries to support HWDSs a feasible and practical solution for widespread deployment? Do software library interfaces reduce or increase the appeal of the HWDS as an abstraction layer? These questions open new directions to explore library and language support for HWDSs. Appendix A describes some initial steps along those directions.

7.6 Hardware Improvements

This thesis focuses on the OS side of the hardware-software interface of HWDSs, and how software improvements along this interface help applications to use HWDSs better. Investigations along the hardware side of the interface may yield benefits for applications as well; this section identifies possible directions to investigate.

7.6.1 Other HWDSs

The most obvious hardware improvement is support for more data structures. What other data structures are amenable to HWDS implementation? Kim [67] identifies the sparse vector and hash table as possibilities for abstract datatype processors, which are closely related to HWDSs. Graphs [85, 41] and trees [117] have been implemented using RC co-processors.

7.6.2 Improved processor pipeline support

This thesis uses a HWDS functional unit that operates atomically and non-speculatively. Since the rate of HWDS instructions usually is slow, these restrictions are not oppressive. However, some heavy uses of HWDSs could be more efficient if the functional unit were able to operate in parallel and speculatively with the rest of the pipeline, and if the functional unit itself could be pipelined. An example of such a use is in overflow handling and context switching, which execute repeated SPILL or FILL instructions.

7.6.3 HWDS support for instructions

The HWDSs presented in this thesis implement the SPILL and FILL instructions with combinations of other instructions, real and imagined. What if the HWDS supported SPILL and FILL natively? The complexity of OS management would lessen, since the interface to the HWDS would be cleaner, albeit slightly larger. More important, the hardware would be responsible for providing the most efficient mechanisms for getting data in and out. One step further, perhaps the hardware can implement `SPILL` and `FILL` to access memory directly. Then the processor can be freed to do other work until the HWDS is finished. Such support would permit HWDS events such as context switching and overflow handling to work asynchronously and hide much of the overhead induced by those events.

Another intriguing possibility for HWDS instructions is for the hardware to convert failover operations directly into overflow data structure operations. Such conversion would permit failover to happen asynchronously, permitting the processor or HWDS to do other work. Such a hardware improvement is reminiscent of stored microprograms [75, 86] and instruction fusion [29].

Yet another mechanism for efficient filling would be to use the idea of *paired operations* proposed by Leiserson [77]. An example of a paired operation is a [`DEQUEUE`, `ENQUEUE`]; when overflow exists, a `DEQUEUE` can be paired implicitly with an `ENQUEUE` from the overflow nodes. In a priority queue that only reads from the head of the queue, such a paired operation has the potential to eliminate underflow.

7.6.4 Prefetching

Prefetching is known to decrease (and sometimes increase) cache miss rates. The structural locality embedded in a HWDS seems perfect for implementing a linked prefetcher. The HWDSs used in this thesis only store key-value pairs; in practice, values are likely pointers to structured data that an application uses. Prefetch logic could load the data pointed to by the values for nodes in the HWDS. Evaluations for prefetching support necessarily must consider the cost of prefetching, which is increased memory bus pressure and the possibility of increased miss rates due to cache evictions caused by prefetched cache lines.

7.6.5 Multicore considerations

This thesis considered the integration of HWDSs in a uniprocessor computer architecture. Modern systems increasingly rely on multiprocessing, in particular chip multicore multiprocessing, which warrants further investigations into how HWDSs should be accessed by hardware. Similar problems as caching—coherency, scalability, sharing, and hierarchy—may appear in such investigations. The combination of data sharing and multicore is appealing to study with HWDSs; multicore processors increase contention on shared data, and if a HWDS can manage contention better than alternatives such as locking and TM, then the HWDS may have an even greater benefit to multicore than to single core computers. Multicore warrants further study of HWDSs.

7.7 Conclusion

This thesis ponders: How should computers access memory? Since memory latency improves slower than bandwidth, which improves slower than processor speed, memory accesses hamper computer system performance. Although caching alleviates some of the latency problems, when the cache inevitably misses, performance suffers. Instead of operating in terms of memory (cache) accesses, this thesis argues that computer architecture and operating systems cooperate to support programming with data structure operations, the common coin of modern programming languages.

Bibliography

[1] Trees II: red-black trees [LWN.net]. http://lwn.net/Articles/184495/, 2006.

[2] Boost c++ libraries. http://www.boost.org/, 2012.

[3] FlightGear. http://www.flightgear.org/, 2012.

[4] Geant4: A toolkit for the simulation of the passage of particles through matter. http://geant4.cern.ch/, 2012.

[5] gem5. http://www.m5sim.org/Main_Page, 2012.

[6] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: programming model, algorithms and evaluation. *SIGPLAN Not.*, 33(11):8191, October 1998.

[7] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, page 5166, Berkeley, CA, USA, 2009. USENIX Association.

[8] Ghiath Al-Kadi and Andrei Terechko. A hardware task scheduler for embedded video processing. In *High Performance Embedded Architectures and Compilers*, pages 140–152. 2009.

[9] J. Allison, K. Amako, J. Apostolakis, H. Araujo, P.A. Dubois, M. Asai, G. Barrand, R. Capra, S. Chauvie, R. Chytracek, G.A.P. Cirrone, G. Cooperman, G. Cosmo, G. Cuttone, G.G. Daquino, M. Donszelmann, M. Dressel, G. Folger, F. Foppiano, J. Generowicz, V. Grichine, S. Guatelli, P. Gumplinger, A. Heikkinen, I. Hrivnacova, A. Howard, S. Incerti, V. Ivanchenko, T. Johnson, F. Jones, T. Koi, R. Kokoulin, M. Kossov, H. Kurashige, V. Lara, S. Larsson, F. Lei, O. Link, F. Longo,

M. Maire, A. Mantero, B. Mascialino, I. McLaren, P.M. Lorenzo, K. Minamimoto, K. Murakami, P. Nieminen, L. Pandola, S. Parlati, L. Peralta, J. Perl, A. Pfeiffer, M.G. Pia, A. Ribon, P. Rodrigues, G. Russo, S. Sadilov, G. Santin, T. Sasaki, D. Smith, N. Starkov, S. Tanaka, E. Tcherniaev, B. Tome, A. Trindade, P. Truscott, L. Urban, M. Verderi, A. Walkden, J.P. Wellisch, D.C. Williams, D. Wright, and H. Yoshida. Geant4 developments and applications. *IEEE Transactions on Nuclear Science*, 53(1):270 –278, February 2006.

[10] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), page 483485, New York, NY, USA, 1967. ACM. ACM ID: 1465560.

[11] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284 –292, September 1993.

[12] G. J. Battarel and R. J. Chevance. Design of a high level language machine. *SIGARCH Comput. Archit. News*, 6(9):5–17, 1978.

[13] Jim Bell and Gopal Gupta. An evaluation of selfadjusting binary search tree techniques. *Software: Practice and Experience*, 23(4):369–382, April 1993.

[14] R. Bhagwan and B. Lin. Fast and scalable priority queue architecture for high-speed network switches. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 538–547 vol.2, 2000.

[15] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):17, August 2011.

[16] Gedare Bloom, Gabriel Parmer, Bhagirath Narahari, and Rahul Simha. Real-time scheduling with hardware data structures. In *Work-in-Progress Session. IEEE Real-Time Systems Symposium*, December 2010.

[17] Gedare Bloom, Gabriel Parmer, Bhagirath Narahari, and Rahul Simha. Shared hardware data structures for hard real-time systems. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '12, page 133142, New York, NY, USA, 2012. ACM.

[18] J. Bobba, N. Goyal, M.D. Hill, M.M. Swift, and D.A. Wood. TokenTM: efficient execution of large transactions with hardware transactional memory. In *35th International Symposium on Computer Architecture, 2008. ISCA '08*, pages 127 –138, June 2008.

[19] Haran Boral and David J. DeWitt. Parallel architectures for database systems. page 1128. IEEE Press, Piscataway, NJ, USA, 1989.

[20] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, page 18, Berkeley, CA, USA, 2010. USENIX Association.

[21] Jay B. Brockman, Shyamkumar Thoziyoor, Shannon K. Kuntz, and Peter M. Kogge. A low cost, multithreaded processing-in-memory system. In *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture*, WMPI '04, page 1622, New York, NY, USA, 2004. ACM.

[22] Gerth Stølting Brodal. Worst-case efficient priority queues. In *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, SODA '96, page 5258, Atlanta, Georgia, United States, 1996. Society for Industrial and Applied Mathematics. ACM ID: 313883.

[23] Ioana Burcea, Livio Soares, and Andreas Moshovos. Pointy: a hybrid pointer prefetcher for managed runtime systems. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, page 97106, New York, NY, USA, 2012. ACM.

[24] Wayne Burleson, Jason Ko, Douglas Niehaus, Krithi Ramamritham, John A. Stankovic, Gary Wallace, and Charles Weems. The spring scheduling coprocessor: a scheduling accelerator. *IEEE Trans. Very Large Scale Integr. Syst.*, 7(1):38–47, 1999.

[25] Robert D. Cameron and Dan Lin. Architectural support for SWAR text processing with parallel bit streams: the inductive doubling principle. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 337–348, Washington, DC, USA, 2009. ACM.

[26] Center for Discrete Mathematics & Theoretical Computer Science. 9th DIMACS implementation challenge: Shortest paths. http://www.dis.uniroma1.it/challenge9/download.shtml, 2012.

[27] R. Chandra and O. Sinnen. Improving application performance with hardware data structures. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1 –4, April 2010.

[28] Bernard Chazelle. The soft heap: an approximate priority queue with optimal error rate. *J. ACM*, 47(6):10121027, November 2000. ACM ID: 355554.

[29] Allen C. Cheng. Amplifying embedded system efficiency via automatic instruction fusion on a post-manufacturing reconfigurable architecture platform. In *Quality Electronic Design, International Symposium on*, pages 744–749, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[30] Y.H. Cho and W.H. Mangione-Smith. Deep packet filter with dedicated logic and read only memories. In *FCCM 2004*, 2004.

[31] Slo-Li Chu and Tsung-Chuan Huang. SAGE: an automatic analyzing system for a new high-performance SoC architecture-processor-in-memory. *J. Syst. Archit.*, 50(1):115, January 2004.

[32] Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie, and Dan Grossman. ASF: AMD64 extension for lock-free data structures and transactional memory. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, page 3950, Washington, DC, USA, 2010. IEEE Computer Society.

[33] Nathan Clark, Amir Hormati, and Scott Mahlke. VEAL: virtualized execution accelerator for loops. In *ACM SIGARCH Computer Architecture News*, ISCA '08, page 389400, Washington, DC, USA, 2008. IEEE Computer Society. ACM ID: 1382155.

[34] Ellis Cohen and David Jefferson. Protection in the hydra operating system. *SIGOPS Oper. Syst. Rev.*, 9(5):141–160, 1975.

[35] Robert P. Colwell, Edward F. Gehringer, and E. Douglas Jensen. Performance effects of architectural complexity in the intel 432. *ACM Trans. Comput. Syst.*, 6(3):296339, August 1988.

[36] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. *SIGARCH Comput. Archit. News*, 30(5):279290, October 2002.

[37] George W. Cox, William M. Corwin, Konrad K. Lai, and Fred J. Pollack. Interprocess communication and processor dispatching on the intel 432. *ACM Trans. Comput. Syst.*, 1(1):4566, February 1983.

[38] Alberto R. Cunha, Carlos N. Ribeiro, and Jos A. Marques. The architecture of a memory management unit for object-oriented systems. *SIGARCH Comput. Archit. News*, 19(4):109116, July 1991.

[39] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. *SIGPLAN Not.*, 46(3):3952, March 2011.

[40] A. DeHon, J. Adams, M. DeLorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. Vanier, and M. Wrighton. Design patterns for reconfigurable computing. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 13 – 23, April 2004.

[41] Michael deLorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomas E. Uribe, Thomas F. Jr Knight, and Andre DeHon. Graph-Step: a system architecture for sparse-graph algorithms. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 143151, Washington, DC, USA, 2006. IEEE Computer Society. ACM ID: 1170448.

[42] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, 1966.

[43] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 157–168, Washington, DC, USA, 2009. ACM.

[44] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. The architecture of the DIVA processing-in-memory chip. In *Proceedings of the 16th international conference on Supercomputing*, ICS '02, page 1425, New York, NY, USA, 2002. ACM.

[45] E. Ebrahimi, O. Mutlu, and Y.N. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *IEEE 15th International Symposium on High Performance Computer Architecture, 2009. HPCA 2009*, pages 7 –17, February 2009.

[46] Zhen Fang, Lixin Zhang, John B. Carter, Ali Ibrahim, and Michael A. Parker. Active memory operations. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, page 232241, New York, NY, USA, 2007. ACM.

[47] Basilio B. Fraguela, Jose Renau, Paul Feautrier, David Padua, and Josep Torrellas. Programming the FlexRAM parallel intelligent memory system. *SIGPLAN Not.*, 38(10):4960, June 2003.

[48] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1-4):111–129, November 1986.

[49] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596615, July 1987. ACM ID: 28874.

[50] Free Software Foundation. The GNU c++ library. http://gcc.gnu.org/onlinedocs/libstdc++/, 2012.

[51] Dan Gibson. *Scalable Cores in Chip Multiprocessors.* PhD thesis, University of Wisconsin-Madison, 2010.

[52] Dan Gibson and David A Wood. Forwardflow: a scalable core for power-constrained CMPs. In *ACM SIGARCH Computer Architecture News*, volume 38, page 1425, New York, NY, USA, June 2010. ACM. ACM ID: 1815966.

[53] Haiku, Inc. Haiku project. http://www.haiku-os.org/, 2012.

[54] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289300, May 1993.

[55] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41:3338, July 2008. ACM ID: 1449387.

[56] J Hoogerbrugge and A Terechko. A multithreaded multicore system for embedded media processing. *Transactions on High-Performance Embedded Architectures and Compilers*, 3(2), 2008.

[57] Christopher J. Hughes and Sarita V. Adve. Memory-side prefetching for linked data structures for processor-in-memory systems. *Journal of Parallel and Distributed Computing*, 65(4):448–463, April 2005.

[58] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 35, page 186197, New York, NY, USA, June 2007. ACM. ACM ID: 1250686.

[59] Prabhat Jain, G. Edward Suh, and Srinivas Devadas. Embedded intelligent SRAM. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, page 869874, New York, NY, USA, 2003. ACM.

[60] Weirong Jiang, Yi-Hua E. Yang, and Viktor K. Prasanna. Scalable multi-pipeline architecture for high performance multi-pattern string matching. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12, Atlanta, GA, USA, 2010.

[61] Douglas W Jones. An empirical comparison of priority-queue and event-set implementations. *Commun. ACM*, 29(4):300311, April 1986. ACM ID: 5686.

[62] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, page 252263, New York, NY, USA, 1997. ACM.

[63] Y Kawanaka, S Wakabayashi, and S Nagayama. A systolic regular expression pattern matching engine and its application to network intrusion detection. Taipei, 2008.

[64] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Rec.*, 27(3):4252, September 1998.

[65] Byung Kook Kim and K.G. Shin. Scalable hardware earliest-deadline-first scheduler for ATM switching networks. In *Real-Time Systems Symposium, IEEE International*, page 210, Los Alamitos, CA, USA, 1997. IEEE Computer Society.

[66] Changkyu Kim, Simha Sethumadhavan, M. S Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W Keckler. Composable lightweight processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, page 381394, Washington, DC, USA, 2007. IEEE Computer Society. ACM ID: 1331733.

[67] Martha Kim. Stories, not words: Abstract datatype processors. In *Workshop on New Directions in Computer Architecture (NDCA)*, 2011.

[68] Donald E Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching.* Addison Wesley Longman Publishing Co., Inc., 1998.

[69] Paul Kohout, Brinda Ganesh, and Bruce Jacob. Hardware support for real-time operating systems. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 45–51, Newport Beach, CA, USA, 2003. ACM.

[70] Christos Kozyrakis and David Patterson. Overcoming the limitations of conventional vector processors. In *ACM SIGARCH Computer Architecture News*, volume 31, page 399409, New York, NY, USA, May 2003. ACM. ACM ID: 859664.

[71] Pramote Kuacharoen, Mohamed A Shalan, and Vincent J. Mooney III. A configurable hardware scheduler for real-time systems. *In Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 96—101, 2003.

[72] Chetan Kumar, Sudhanshu Vyas, Jonathan Shidal, Ron Cytron, Christopher Gill, Joseph Zambreno, and Phillip Jones. Improving system predictability and performance via hardware accelerated data structures, 2012.

[73] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 162–173, San Diego, California, USA, 2007. ACM.

[74] Anthony LaMarca and Richard Ladner. The influence of caches on the performance of heaps. *J. Exp. Algorithmics*, 1, January 1996. ACM ID: 235145.

[75] James R Larus. A comparison of microcode, assembly code, and high-level languages on the VAX-11 and RISC i. *ACM SIGARCH Computer Architecture News*, 10:1015, September 1982. ACM ID: 641561.

[76] Sanghoon Lee, Davesh Tiwari, Yan Solihin, and James Tuck. HAQu: hardware accelerated queueing for fine-grained threading on a chip multiprocessor. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture*, 2011.

[77] Charles E. Leiserson. Systolic priority queues. In *Proceedings of the Caltech Conference on Very Large Scale Integration*, pages 199–214, Pasadena, CA, January 1979.

[78] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, 1984.

[79] Chyuan Shiun Lin, Diane C. P. Smith, and John Miles Smith. The design of a rotating associative memory for relational database applications. *ACM Trans. Database Syst.*, 1(1):5365, March 1976.

[80] Lixia Liu and Silvius Rus. Perflint: A context sensitive performance advisor for c++ programs. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, page 265274, Washington, DC, USA, 2009. IEEE Computer Society.

[81] J. Loew, J. Elwell, D. Ponomarev, and P.H. Madden. A co-processor approach for accelerating data-structure intensive algorithms. In *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 431–438, 2010.

[82] Cris L Luengo Hendriks. Revisiting priority queues for image analysis. *Pattern Recogn.*, 43(9):30033012, September 2010. ACM ID: 1808374.

[83] Milo M. K Martin, Daniel J Sorin, Bradford M Beckmann, Michael R Marty, Min Xu, Alaa R Alameldeen, Kevin E Moore, Mark D Hill, and David A Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33:9299, November 2005. ACM ID: 1105747.

[84] Austen McDonald, Brian D. Carlstrom, JaeWoong Chung, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Transactional memory: The hardware-software interface. *IEEE Micro*, 27(1):6776, 2007.

[85] Oskar Mencer, Zhining Huang, and Lorenz Huelsbergen. HAGAR: efficient multi-context graph processors. In *Proceedings of the Reconfigurable Computing Is Going*

Mainstream, 12th International Conference on Field-Programmable Logic and Applications, FPL '02, page 915924, London, UK, UK, 2002. Springer-Verlag. ACM ID: 740064.

[86] M.V. Milkes. The genesis of microprogramming. *IEEE Annals of the History of Computing*, 8(2):116–126, 1986.

[87] S.-W. Moon, K.G. Shin, and J. Rexford. Scalable hardware priority queue architectures for high-speed packet switches. In *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, pages 203–212, 1997.

[88] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. LogTM: log-based transactional memory. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006*, pages 254 – 265, February 2006.

[89] A. Morton, J. Liu, and Insop Song. Efficient priority-queue data structure for hardware implementation. In *International Conference on Field Programmable Logic and Applications, 2007. FPL 2007*, pages 476 –479, August 2007.

[90] G. J. Myers and B. R. S. Buckingham. A hardware implementation of capability-based addressing. *SIGOPS Oper. Syst. Rev.*, 14(4):13–25, 1980.

[91] OAR Corporation. RTEMS: real-time executive for multiprocessor systems. http://www.rtems.com/, 2012.

[92] Oracle. MySQL :: The world's most popular open source database. http://www.mysql.com/, 2012.

[93] Elliot I. Organick. *A programmer's view of the Intel 432 system.* McGraw-Hill, Inc., New York, NY, USA, 1983.

[94] M. Oskin, F.T. Chong, and T. Sherwood. ActiveOS: virtualizing intelligent memory. In *(ICCD '99) International Conference on Computer Design, 1999*, pages 202 –208, 1999.

[95] Mark Oskin, Frederic T Chong, and Timothy Sherwood. Active pages: a computation model for intelligent memory. In *ACM SIGARCH Computer Architecture News*, ISCA '98, page 192203, Washington, DC, USA, 1998. IEEE Computer Society. ACM ID: 279387.

[96] E. A. Ozkarahan, S. A. Schuster, and K. C. Smith. RAP: an associative processor for data base management. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, AFIPS '75, page 379387, New York, NY, USA, 1975. ACM.

[97] K. Pagiamtzis and A. Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: a tutorial and survey. *Solid-State Circuits, IEEE Journal of*, 41(3):712–727, 2006.

[98] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, 1997.

[99] David A Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):7175, October 2004. ACM ID: 1022596.

[100] Ben Pfaff. Performance analysis of BSTs in system software. *SIGMETRICS Perform. Eval. Rev.*, 32(1):410411, June 2004. ACM ID: 1005742.

[101] Isabelle Puaut and Christophe Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proceedings of the conference on*

Design, automation and test in Europe, DATE '07, page 14841489, San Jose, CA, USA, 2007. EDA Consortium.

[102] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668676, June 1990.

[103] ReactOS Foundation. ReactOS. http://www.reactos.org/en/index.html, 2012.

[104] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, page 6273, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[105] Robert Rönngren and Rassul Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Trans. Model. Comput. Simul.*, 7(2):157209, April 1997.

[106] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. TxLinux: using and managing hardware transactional memory in an operating system. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 87–102, Stevenson, Washington, USA, 2007. ACM.

[107] Amir Roth and Gurindar S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th annual international symposium on Computer architecture*, ISCA '99, page 111121, Washington, DC, USA, 1999. IEEE Computer Society.

[108] S. Saez, J. Vila, A. Crespo, and A. Garcia. A hardware scheduler for complex real-time systems. In *Industrial Electronics, 1999. ISIE '99. Proceedings of the IEEE International Symposium on*, volume 1, pages 43–48 vol.1, 1999.

[109] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 311–322, Pittsburgh, Pennsylvania, USA, 2010. ACM.

[110] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W Keckler, and Charles R Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *ACM SIGARCH Computer Architecture News*, volume 31, page 422433, New York, NY, USA, May 2003. ACM. ACM ID: 859667.

[111] David Elliot Shaw. A relational database machine architecture. In *Proceedings of the fifth workshop on Computer architecture for non-numeric processing*, CAW '80, page 8495, New York, NY, USA, 1980. ACM.

[112] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Database-aware semantically-smart storage. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, page 1818, Berkeley, CA, USA, 2005. USENIX Association.

[113] M. Sjalander, A. Terechko, and M. Duranton. A look-ahead task management unit for embedded multi-core architectures. In *Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on*, pages 149–157, 2008.

[114] D.L. Slotnick. Logic per track devices. In Franz L. Alt and Morris Rubinoff, editors, *Advances in Computers*, volume Volume 10, pages 291–296. Elsevier, 1970.

[115] Insop Song. HyOS: a hybrid operating system design approach for real-time systems using hardware acceleration. Chapel Hill, North Carolina, 2012. Open Source Automation Development Lab.

[116] Standard Performance Evaluation Corporation. SPEC CPU2006. http://www.spec.org/cpu2006/, 2012.

[117] Song Sun and Joseph Zambreno. Design and analysis of a reconfigurable platform for frequent pattern mining. *IEEE Transactions on Parallel and Distributed Systems*, 22(9):1497–1505, September 2011.

[118] Lin Tan and Timothy Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *ACM SIGARCH Computer Architecture News*, volume 33, page 112122, New York, NY, USA, May 2005. ACM. ACM ID: 1069981.

[119] Justin Teller, Charles B. Silio Jr, and Bruce Jacob. Performance characteristics of MAUI: an intelligent memory system architecture. In *Proceedings of the 2005 workshop on Memory system performance*, MSP '05, page 4453, New York, NY, USA, 2005. ACM.

[120] The Battle for Wesnoth. Battle for wesnoth. http://www.wesnoth.org/, 2012.

[121] The Chromium Project. Chromium. http://www.chromium.org/Home, 2012.

[122] The Document Foundation. Home LibreOffice. http://www.libreoffice.org/, 2012.

[123] Mikkel Thorup. Equivalence between priority queues and sorting. *J. ACM*, 54(6), December 2007. ACM ID: 1314692.

[124] Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael M. Swift, and Adam Welc. xCalls: safe I/O in memory transactions. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, page 247260, New York, NY, USA, 2009. ACM.

[125] Qing Wan, Hui Wu, and Jingling Xue. WCET-aware data selection and allocation for scratchpad memory. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, page 4150, New York, NY, USA, 2012. ACM.

[126] Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. Guided region prefetching: a cooperative hardware/software approach. *SIGARCH Comput. Archit. News*, 31(2):388398, May 2003.

[127] Yasuko Watanabe, John D Davis, and David A Wood. WiDGET: wisconsin decoupled grid execution tiles. In *ACM SIGARCH Computer Architecture News*, volume 38, page 213, New York, NY, USA, June 2010. ACM. ACM ID: 1815965.

[128] J. Whitham and N. Audsley. Investigating average versus worst-case timing behavior of data caches and data scratchpads. In *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, pages 165 –174, July 2010.

[129] J. Whitham and N. Audsley. Studying the applicability of the scratchpad memory management unit. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 205 –214, April 2010.

[130] Maurice V Wilkes. The memory gap and the future of high performance memories. *SIGARCH Comput. Archit. News*, 29(1):27, March 2001. ACM ID: 373576.

[131] Hugh E Williams, Justin Zobel, and Steffen Heinz. Self-adjusting trees in practice for large text collections. *Software: Practice and Experience*, 31(10):925–939, August 2001.

[132] Willow Garage. OpenCV. http://opencv.willowgarage.com/wiki/, 2012.

[133] Hui Wu, Jingling Xue, and Sri Parameswaran. Optimal WCET-aware code selection for scratchpad memory. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, page 5968, New York, NY, USA, 2010. ACM.

[134] L. Wu, M. Kim, and S. Edwards. Cache impacts of datatype acceleration. *IEEE Computer Architecture Letters*, 11(1):21–24, January 2012.

[135] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: the kernel of a multiprocessor operating system. *Commun. ACM*, 17(6):337–345, 1974.

[136] Wm. A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23:2024, March 1995. ACM ID: 216588.

[137] Chia-Lin Yang and Alvin R. Lebeck. A programmable memory hierarchy for prefetching linked data structures. In *Proceedings of the 4th International Symposium on High Performance Computing*, ISHPC '02, page 160174, London, UK, UK, 2002. Springer-Verlag.

[138] Chia-Lin Yang, Alvin R. Lebeck, Hung-Wei Tseng, and Chien-Hao Lee. Tolerating memory latency through push prefetching for pointer-intensive applications. *ACM Trans. Archit. Code Optim.*, 1(4):445475, December 2004.

[139] Yi-Hua Edward Yang and Viktor K. Prasanna. Memory-efficient pipelined architecture for large-scale string matching. In *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, pages 104–111, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[140] Junhee Yoo, Sungjoo Yoo, and Kiyoung Choi. Multiprocessor system-on-chip designs with active memory processors for higher memory efficiency. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 806–811, 2009.

Appendix A – STL Profiling: Containers and Comparators

This appendix describes profiling the STL to determine how applications use containers and object comparisons.

A.1 Maps in the C++ STL

Linear and tree-based structures, which underlie STL containers, are commonly used by programmers. Commonly used containers are the vector, list, set, and map; these containers are included by at least half of 21 open-source C++ programs covering a range of application domains including navigation, simulation, computer vision, video games, document processing, databases, operating systems, and web browsers. Table A-1 shows how many of the following programs include which STL header files: dimacs-sq, Dijkstra's algorithm with SmartQ [26]; Opal [83] and GEM5 [15, 5], processor simulators; Geant4 [9, 4], physics particle simulator; FlightGear [3], flight simulator; Wesnoth [120], video game; OpenCV [132], computer vision library; Boost [2], library for C++; MySQL [92], database server; LibreOffice [122], office productivity suite; Doxygen, documentation generation; Haiku [53], OS based on BeOS; ReactOS [103], OS based on Windows NT; Chromium [121], web browser; povray, soplex, dealII, namd, xalancbmk, astar, and omnetpp, C++ benchmarks from SPEC CPU 2006 [116].

Digging deeper, I investigated the runtime behavior of two programs—Geant4 and Chromium—that rely heavily on the STL containers. I modified the profile mode of the

Table A-1: STL container use of 21 open-source C++ programs.

vector	list	set	multiset	map	bitset	unordered set	unordered map
13	13	13	1	14	6	4	5

Table A-2: STL map use profiling. Map count is the number of maps created by the application. Find time is the percent of application execution spent executing find operations. Max find is the size and find time for the map with the most time spent executing find.

Application	Map count	Map size		Find time	Max find	
		Mean	Max		Map size	% Time
Chromium	1907	5.33	1228	12.39%	254	3.55%
Geant4: DNAPhysics	126	3074.59	88680	23.43%	38	11.38%
gammaray_telescope	750	73.87	466	8.78%	466	0.24%
hadrontherapy	752	74.30	467	5.07%	467	0.15%
human_phantom	72	3459.30	21683	5.08%	408	4.41%
microbeam	540	529.89	21683	3.37%	240	0.74%
brachytherapy	67	3744.83	21683	3.09%	576	2.43%
radioprotection	771	239.08	53256	2.69%	488	0.06%
lAr_calorimeter	751	74.83	467	1.90%	467	0.07%
medical_linac	55	4544.80	21683	1.02%	224	0.71%
underground_physics	58	4297.58	21683	0.90%	312	0.80%

GNU C++ Library [50], which is based on the Perflint [80] project. I added detailed statistics for the map and vector containers including the minimum, maximum, and total number of elements, and the time of insert, erase, and find operations. Geant4 and Chromium are built and executed with and without profile mode support. For Geant4, the advanced examples provided with the release are used as a workload. For Chromium, the workload was just to start the browser—which loads a blank page by default—and then close the browser interactively; this workload is subject to timing variations and is only useful for descriptive empirical evidence.

The profiled version of each application is executed to obtain measurements for the STL container operations, and the unprofiled version is executed to obtain a measure for the overall workload execution time without the overhead of profiling. Table A-2 shows the results of these runs with the Chromium and Geant4 workloads.

Chromium spends about 12% of its time executing find for the simple task of starting up and shutting down, and the Geant4 DNAPhysics example spends over 23% of its execution time on find and around 11% of its execution time searching through a map with only 38 elements. Such a map is an ideal candidate for hardware acceleration: small and frequently accessed. Even in Chromium, a map of only 254 elements consumes roughly 3.5% of execution time.

A.2 Object comparison code

Using HWDSs with objects is challenging for the STL set and map, because programmers can write custom key and value comparison code, which can be hard to support with parallel hardware comparators; performance benefits of HWDSs comes especially from parallel comparisons. Efficient comparison hardware exists for primitive data types including integers, floats, and strings. If object-oriented programs use other data types, or complicated combinations of these primitives, then HWDSs would have difficulty providing any benefits. This section describes a cursory investigation into whether C++ programs use complicated comparisons or simple, supported primitives.

To get a sense of whether C++ programs use complex comparisons with containers, I extended the profile mode (used in Section A.1) with support for printing the call site of container instantiation. The call site gives the code location where an object instance is made, and whether it uses a primitive comparison (i.e. int, float, or string), or if the structure has some alternate comparison method. Using this profiling information, the behavior of the heaviest usage of the STL map container in the two profiled applications—Chromium and Geant4—can be found. The most heavily used maps in both applications are maps that make straightforward comparisons with primitives (integer and double). The other maps tend to use integer, floating point, or string comparisons.

The map that consumes the most time for the Chromium workload is the `observers_` map declared in the `NotificationServiceImpl` class and used for event notification, in particular for tracking observers of notifications. This map has integer keys with another map (the event sources and observers) as its values; Figure A-1 shows the definition of the map. The use of integer keys makes it viable for a map HWDS, as does its small maximum size of 254 elements. The map that consumes the most time in the DNAPhysics example of Geant4 uses `double` keys (and has another map has its value). By observation, the map keys that are used in the Geant4 code base are integers, doubles, and strings; other maps have key types that are obscured by classes and type definitions.

```
class NotificationServiceImpl
  : public content::NotificationService {
...
typedef ObserverList<content::NotificationObserver> NotificationObserverList;
typedef std::map<uintptr_t, NotificationObserverList*> NotificationSourceMap;
typedef std::map<int, NotificationSourceMap> NotificationObserverMap;
...
NotificationObserverMap observers_;
};
```

Figure A-1: The `observers_` map consumes 3.5% of the Chromium workload's execution time and uses a primitive type (integers) for its key.

A.3 Summary

This appendix shows that real-world applications use STL maps in ways that are amenable to HWDS support.