Shared Hardware Data Structures for Hard Real-Time Systems

Gedare Bloom
Dept. of Computer Science
George Washington University
Washington, DC 20052
gedare@gwu.edu

Gabriel Parmer
Dept. of Computer Science
George Washington University
Washington, DC 20052
gparmer@gwu.edu

Bhagirath Narahari
Dept. of Computer Science
George Washington University
Washington, DC 20052
narahari@gwu.edu

Rahul Simha
Dept. of Computer Science
George Washington University
Washington, DC 20052
simha@gwu.edu

ABSTRACT

Hardware support can reduce the time spent operating on data structures by exploiting circuit-level parallelism. Such hardware data structures (HWDSs) can reduce the latency and jitter of data structure operations, which can benefit real-time systems by reducing worst-case execution times (WCETs). For example, a hardware priority queue (HWPQ) can enqueue and dequeue prioritized items in constant time with low variance; the best software implementations are in logarithmic-time asymptotic complexity for at least one of the enqueue or dequeue operations. The main problems with HWDSs are the limited size of hardware and the complexity of sharing it. In this paper we show that software support can help circumvent the size and sharing limitations of hardware so that applications can benefit from a HWDS. We evaluate our work by showing how the choice of software or hardware affects schedulability of task sets that use multiple priority queues of varying sizes. We model task behavior on 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. Our 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.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Real-time systems and embedded systems; B.3.3 [**Memory Structures**]: Worst-case analysis; E.1 [**Data Structures**]

General Terms

Algorithms, Performance

Keywords

Hardware data structures, schedulability, priority queue

1. INTRODUCTION

Throughout the history of computing there has been a performance gap between CPUs and main memory. Wilkes [28] points out that memory started out different from and underperforming processing, and the performance gap persisted despite the use of semiconductors for both since the 1970s. Indeed, the performance gap has steadily increased since the 1980s, leading Wulf and McKee [30] 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.

A common technique to delay the impact of the memory wall is caching. Unfortunately data caches are difficult to model in hard real-time systems; in particular obtaining an accurate worst-case execution time (WCET) is hard [8]. One well-known approach to dealing with the data cache is to attempt to partition it among the system's tasks [24]. However data sharing and RTOS services can make cache partitioning problematic and workarounds lead to locking or other strategies that frustrate real-time analysis. (The instruction cache is easier to handle and well-known techniques can accommodate it in a WCET analysis [20].)

Another common approach to reducing the effect of memory latency is to reduce the complexity and operating frequency of processing cores while maintaining high throughput by replicating multiple cores on one chip: the chip multiprocessor or multicore. Multicore platforms have become common due to the continued growth of chip space predicted by Moore's law. For real-time systems, problems with using multicore platforms include the difficulties of parallelizing applications, managing shared caches, partitioning tasks across cores, and synchronizing shared resources among cores; these problems drive up the complexity (and therefore cost) to develop real-time systems using multicore platforms. We suggest an alternative approach for using the spare chip space.

In this paper we introduce hardware data structures (HWDSs) as an approach for hard real-time systems to improve the predictability of memory accesses. HWDSs represent an alternate use of chip space compared to replicating processing cores. The devotion of chip space to HWDSs is a promising direction for time-predictability in hard real-time systems. The contributions of this work include: fitting HWDSs into traditional response time analysis by identifying variables that affect WCET when using a HWDS; deriving two novel assignment algorithms for choosing which tasks should use a HWDS as opposed to a software DS; exploring the parameter space of the variables that affect HWDS WCET in order to quantify the effectiveness of those algorithms; and demonstrating how real-world applications can benefit from this work.

The advantage that HWDS have over alternatives such as scratchpad memory is that a HWDS exploits hardware parallelism to reduce the algorithmic complexity of data structure operations while also improving memory access predictability. The primary disadvantage for HWDSs is that chip designers need to devote a fixed size of hardware resources for the benefit of DS operations, and the limited size causes problems both for sharing the hardware and for using the HWDS when DS size exceeds HWDS capacity. We circumvent hardware size constraints by using exceptions triggered in hardware and handled in software; amortizing exception costs across multiple non-excepting operations puts bounds on the cost of HWDS operations for schedulability analysis. For sharing a HWDS we present two simple algorithms and two novel algorithms that determine which tasks and data structures can use HWDS resources subject to task, data structure, and HWDS parameters.

2. HARDWARE DATA STRUCTURES

A *hardware data structure* (HWDS) is an implementation of a data structure with hardware mechanisms to improve the performance and asymptotic complexity of data structure operations. A HWDS organizes the memory hierarchy in terms of data structure operations instead of cache line fetches. By avoiding the cache a HWDS has the potential to deliver consistent, predictable timing.

So far most work on HWDSs has ignored the interface between the HWDS and programmer, with existing HWDSs having limited interactions with operating system (OS) and application software. This paper shows how such interactions are crucial to realize efficient HWDSs. In particular, we investigate HWDSs from a holistic view that incorporates processor architecture, OS, and applications. OS support extends the capabilities of HWDSs beyond prior art with support for large data structures and sharing a HWDS.

2.1 Priority queue: an example HWDS

A priority queue (PQ) is a data structure with *enqueue* (insert), *dequeue* (delete-min), and *peek* (read first) operations. Dequeue removes and returns the highest priority node in the queue; peek is similar to dequeue, but without removing the node. Applications of PQs include graph problems like finding the minimum spanning tree or shortest path, discrete event simulation [9], network routing [19], OS scheduling [5], and image analysis [16].

Although many software implementations of PQs exist, the implicit binary heap remains one of the best due to its simplicity, logarithmic worst-case time complexity, and low memory overhead [15, 16]; these points especially are valid in real-time systems [18]. In this paper we consider only the implicit binary heap as a representative software-implemented PQ.

A hardware priority queue (HWPQ) is a hardware implementation of the PQ data structure. An example of a HWPQ is the shift register PQ, which is shown in Figure 1. The shift register PQ is an array of priority and data payload tuples that the hardware sorts 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 PQ, but each block makes a decision locally based on inputs from its neighbors and the single global input so the design is simple. The latency of this HWPQ primarily comes from the wire delays of the global signals, especially for a

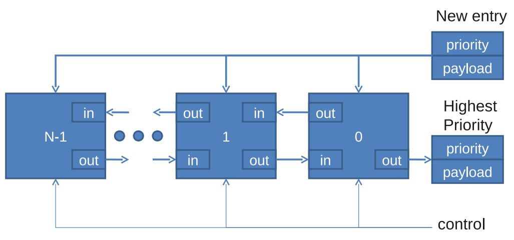

Figure 1: A shift-register based hardware priority queue of priority-payload tuples in a double-linked hardware list.

large number of blocks [19]. Fanout of the comparators is also a concern if there is a wide range of priority values. Other HWPQ implementations eliminate the global lines—see Moon et al. [19]. The choice of HWPQ implementation makes a difference in terms of hardware size, power cost, scalability, and maximum operating frequency, but we defer the reasoning behind such choice.

Both adding and removing nodes with the shift register HWPQ are efficient because priority comparisons occur in parallel and every block determines its action upon receiving global signals. The insert operation 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. Removing the highest priority node is simple, with each block sending its tuple to the right and latching from the left.

HWPQs motivate the HWDS approach. Enqueue and dequeue happen in constant time: the fastest software implementations take logarithmic time for at least some operations. Unfortunately the size constraints and lack of support for sharing the HWPQ among multiple PQs present problems for general-purpose application use of a HWPQ. To address these problems we introduce novel mechanisms for spilling and filling data between the HWPQ and main memory. We start with the established work on fixed-size unshared HWPQs and evolve a new approach that combines reasonable hardware and OS modifications to support application use of a HWPQ.

2.2 Spilling and Filling

Applications require support for PQs of arbitrary size. Since hardware has a fixed capacity, arbitrarily large data sets eventually will cause overflow. In addition, chip space allocated to the HWPQ steals from other features such as cache, so minimizing the HWPQ size is important.

We solve the problem of arbitrarily large PQs using an exception-based approach for handling overflow inspired by work in fine-grained threading [14, 23]. HWPQ control logic and software services handle overflow by spilling HWPQ data to secondary storage (memory). A HWPQ generates an overflow exception when the number of nodes it contains meets some threshold; the maximum threshold is the size of the HWPQ. Similarly, the hardware raises an exception when there exist spilled nodes and either the HWPQ holds less than some threshold of nodes or the highest priority node in the HWPQ has lower priority than some spilled node.

The ordering of nodes in the HWPQ has meaning—based on the interpretation of priority—so the overflow exception handler removes low-priority nodes from the HWPQ. As the exception handler removes nodes the HWPQ marks the lowest priority node remaining in the HWPQ with an *invalid* bit. The hardware will mark invalid any node that the application subsequently enqueues with a lower priority than an existing invalid node. When the head of the HWPQ is invalid the HWPQ raises an underflow exception because a higher priority node might exist among the spilled nodes. The underflow exception handler fills the HWPQ, which mark nodes valid if they have higher priority than those the exception handler fills from the spilled nodes. During an underflow exception the handler may also need to spill nodes because of invalid nodes.

The choice of algorithm for storing the spilled nodes will affect the time required by both the overflow and underflow exception handlers. Because the HWPQ nodes are already sorted we chose to maintain a sorted linked list for the spilled nodes. The overflow handler merge sorts the spilled nodes into the linked list, and the underflow handler fills from the head of the linked list. Other software PQs would likely show advantages for certain PQ sizes, HWPQ sizes, priority value distributions, and PQ access patterns.

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 HWPQ. Exception handler execution time depends on the size of the PQ and the number of nodes spilled (equivalently filled). The rate of exceptions depends on two factors: the rate of PQ operations and the number of nodes spilled. The PQ 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 to be any number k less than or equal to half of the HWPQ size limits the number of exceptions to at most one overflow and one underflow per k PQ operations. If the handler spills 4 nodes then there could be two exceptions for every 4 PQ operations; spilling 8 nodes allows 8 PQ operations with at most two exceptions, and so on. In any window of k PQ operations the worst case is that the entire HWPQ is full of invalid nodes and the HWPQ reads the head node and then enqueues a node. The read induces an underflow exception since the head is invalid. The underflow handler fills the HWPQ with k valid nodes and spills at least k nodes, leaving the HWPQ in a state with at least k valid nodes and possibly invalid nodes filling the rest of the HWPQ. (The HWPQ 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 HWPQ can satisfy at least k operations without another exception. We minimize the number of exceptions that get taken by tuning the handlers to spill half of the PQ size because each exception that gets taken adds extra fixed processing overhead to invoke the handler.

Spilling HWPQ data causes a problem for operations that target spilled nodes: software must implement the operation on the nodes in the spill area. Peek, enqueue, and dequeue operations work fine, but some applications violate the priority abstraction to access PQ nodes at random; for example Dijkstra's algorithm benefits from a change-key operation that can change the priority of an enqueued item, or task schedulers may need to delete a task from the PQ when the task suspends or is killed. Currently we ignore these cases, but we could solve them by introducing an exception to emulate the operation in software; assuming these exceptions are rare or bounded we can analyze them similarly to the overflow and underflow exceptions. For Dijkstra's algorithm change-key can be ignored at the cost of extra storage and processing when dequeueing [7].

2.3 Sharing

Sharing is a traditional OS problem of managing contention for a limited hardware resource. We implement an offline assignment algorithm that decides which task's PQs are allocated the HWPQ. At runtime a HWPQ context switch swaps one PQ for another.

Sharing the HWPQ adds a little more complexity to both hardware and software support. The main addition is that the HW needs to distinguish PQs, so the PQ operations must identify which PQ to use; in prior work there was a one-to-one mapping between PQ and HWPQ. Loosening that mapping to many-to-one introduces the problem that the HWPQ must have some way to separate or distinguish data belonging to different PQs. As with other facets of HWPQ design many possible solutions exist for this problem. Our solution is to add an identifier to every instruction that accesses the HWPQ and for the HWPQ to track which PQ currently is using the HWPQ based on the identifier. The exception handlers use the identifiers to store the spilled nodes for each PQ in a separate data structure. We chose this approach because the hardware cost is small (an extra register and some comparators) while supporting a wide range of policies for how PQs share the HWPQ. The main drawback is that each software PQ must have a unique identifier.

The HWPQ context switch involves spilling all of the nodes for the currently loaded PQ and filling nodes for the PQ the application is accessing. To simplify WCET analysis our HWPQ context switch handler tracks how many nodes it spills from the HWPQ while emptying and refills that PQ so that the same number of nodes are present in the HWPQ before and after a PQ is context switched. We also restrict each task to use at most one PQ; when a task uses more than one PQ only one should use the HWPQ otherwise the number of HWPQ context switches may be large. With our restriction the HWPQ context switch aligns with the task context switch, which is important when analyzing a task's WCET. The worst case cost of a HWPQ context switch is when the HWPQ is full and the handler is refilling from a PQ that had a full HWPQ previously; then the context switch spills and fills the entire HWPQ. Similar to spilling and filling the cost of a HWPQ context switch depends on PQ size and spill data structure implementation.

3. RESPONSE TIME ANALYSIS

A hardware data structure (HWDS) affects task response time by decreasing WCET due to reducing DS operation times, but exceptions caused by overflow/underflow conditions increase WCET. Sharing the HWDS among tasks also increases the response time. In the following we evolve a standard response time analysis [3] to include variables that affect WCET when using a HWDS. We only consider periodic tasks.

We adopt the notation

- τ: the set of all tasks
- T_i: the i'th task
- p_i: period of T_i
- e_i: the worst-case execution time (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.

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

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

Equation 1 considers the WCET of T_i plus the sum of processor time of higher priority tasks overlapping with the time interval t. We find R_i 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 $T_i \in \tau$.

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 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 4).

3.1 Hardware Priority Queues

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

Let $\widehat{S}$ be the size of the HWPQ. 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 PQ 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

$$\widehat{x_i} < A(w) * \lceil O_i/w \rceil. \tag{2}$$

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

$$\widehat{c_i} < A(\widehat{s_i}) * \widehat{s_i}. \tag{3}$$

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 2 and 3 respectively. In Section 5 we measure software and hardware implementations of priority queues for the WCET of their enqueue and dequeue operations and—for HWPQs—context switch, spill, and fill. We evaluate HWDS assignment algorithms with those measurements.

4. HWDS ASSIGNMENT

Assigning the HWDS attempts to assign tasks to use either a HWDS or a software DS. We use terminology from scheduling—indeed the assignment problem is similar to the problem of task scheduling. An assignment is feasible if a solution to Equation 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 we say those tasks are schedulable. An assignment algorithm is optimal if it always produces a feasible assignment for a set of tasks when one exists.

We 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 DS behavior in particular on the WCET of DS operations, HWDS exceptions, and the HWDS context switch time. We established in Section 3.1 that a HWPQ has a bounded WCET if the maximum DS size, maximum number of DS operations per period, and the HWPQ size are bounded. In general these algorithms will work for any HWDS that has bounded WCET based on the DS size and DS operations. If a HWDS requires more information to bound its WCET then new algorithms may be required.

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 DS operations per period the larger HWDSs underperform smaller HWDSs; at such small sizes of DS operations the software DS 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 DS 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.

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 $\widehat{\tau}$,$\widetilde{\tau}$
```

CSCAA (Algorithm 2) is similar to PAA except for the cost heuristic that gets added to the HWDS WCET. We introduce the cost heuristic to penalize low-priority tasks for using the HWDS. 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 we 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. So in Algorithm 2 the function get_cost would return $(c_i - c_m) * (n - i)$ or 0, whichever is greater.

5. EXPERIMENTS

We conducted a series of experiments to evaluate HWDSs in the context of hard real-time systems. These experiments use a HWPQ as an example of a HWDS. We use synthetic task sets to explore the parameter space of the HWPQ as the parameters relate to WCET. We also demonstrate how this work can apply in the real-world by examining the benefits of our approach for workloads approximating real applications.

We implemented a HWPQ within Simics/GEMS [17]—a functionally correct cycle-accurate full system simulator for an out-of-order architecture (based on the ALPHA) that executes the SPARC v9 instruction set. We implemented OS support for HWPQs in the Real-Time Executive for Multiprocessor Systems (RTEMS) [1] open source real-time operating system, which can run on Simics/GEMS. The architectural parameters we chose are representative of an embedded system: 75 MHz CPU, 80 cycle memory latency, and a 4-issue 5-stage pipeline. We extended the SPARC instruction set to support new HWPQ operations directly and added a

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 $\widehat{\tau}$,$\widetilde{\tau}$
```

functional unit to execute the new instructions. This functional unit operates atomically and non-speculatively. Although the HWPQ can achieve single-cycle latencies for PQ operations, restricting the unit to be atomic and non-speculative increases the latency to around 12 cycles for the simulated architectural parameters.

The values we measured for WCET parameters underlie all of the experimental results we present. To estimate the WCET of PQ operations we implemented an implicit binary heap as a representative software PQ. We designed a series of measurement tests that build a PQ up to a specified size and then measure the cost of a PQ operation at that size. We measured five specific events in isolation: enqueue, dequeue, overflow exception, underflow exception, and HWDS context switch. The latter three are only relevant and measured for a HWPQ. We turned off all caching to obtain the WCET of PQ operations. Our approach is pessimistic, but lacking a time-predictable cache leaves few options. As a result the measurements we take are dominated by the memory access latency.

To force the worst-case conditions for the software PQ we measure an enqueue of a node with priority less than the highest-priority node in the heap so that the enqueue operation 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 PQ size). A dequeue of the minimum value causes a maximum amount of work in a heap.

For the HWPQ enqueue and dequeue WCET the HWPQ must be in a state that will not cause an exception. Before measuring enqueue we ensure the HWPQ has enough spare capacity to accept the new node, and before measuring dequeue we ensure at least one valid node is at the head of the HWPQ. To generate the WCET overflow the nodes that get spilled must cause the spill algorithm to do maximum work. We implemented a merge-sorted linked list that iterates from the tail of the spilled nodes to the head (which has highest priority), so to cause the WCET overflow we empty the HWPQ and then fill it with new nodes that have priority less than the head of the linked list. Thus we ensure 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 HWPQ is full of invalid nodes it must fill from the spilled nodes and also spill some of its invalid nodes. We generated the worst-case condition of an underflow by enqueueing nodes with priority less than the head of the spilled nodes (as with the overflow case), invalidated the HWPQ, and then dequeued. The dequeue causes an underflow exception, and the exception handler finds that no capacity exists to fill the HWPQ and so 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. Finally the underflow handler will fill the HWPQ. To cause the WCET of the HWPQ context switch we filled the HWPQ to its maximum size using two separate PQs ensuring the HWPQ contains nodes with priority less than the head of the spilled nodes. Then we cause a HWPQ context switch by issuing an operation for the PQ that is not currently loaded in the HWPQ. The context switch handler spills all of the nodes in the HWPQ which (because of the ordering of nodes) takes maximum time, and then fills the HWPQ with nodes from the next PQs spill data structure.

5.1 Schedulability

To characterize the HWPQ parameter space and evaluate the HWPQ assignment algorithms we designed a series of experiments using synthetic task sets generated as follows. Create a set of n tasks by 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. We use response time analysis (Equation 1) to ensure the generated task set is schedulable, and regenerate any sets that fail the schedulability test.

We then modify each generated task set to include PQ operations parametrized by a max PQ size, max HWPQ size, PQ implementation, and number of operations to complete in a period. Using the task's period and utilization we calculate its compute time and add the WCET determined by the PQ parameters. PQ size and implementation determine the WCET for any given operation and the PQ size with the number of operations determines the WCET for the HWPQ exceptions. The HWPQ and PQ sizes determine the WCET for the HWPQ context switch.

We varied the parameters of max PQ size, PQ implementation, and number of operations in a controlled way. For each particular assignment of parameters we generated 10000 task sets and attempted to assign PQ usage for each task set using all four of the algorithms (SOA, HOA, PAA, and CSCAA) presented in Section 4. For each task set and assignment algorithm we determine whether the task set is schedulable after PQ assignment. For these experiments we set the max HWPQ 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 HWPQ size.

Figure 2 shows the results of our experiments as both the max PQ size and the number of PQ operations per period vary by powers of 2 from 16 to 8192. For this particular figure we set the task set utilization U to 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 PQ assignment for each combination of PQ size and number of PQ operations. We also plot a line 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

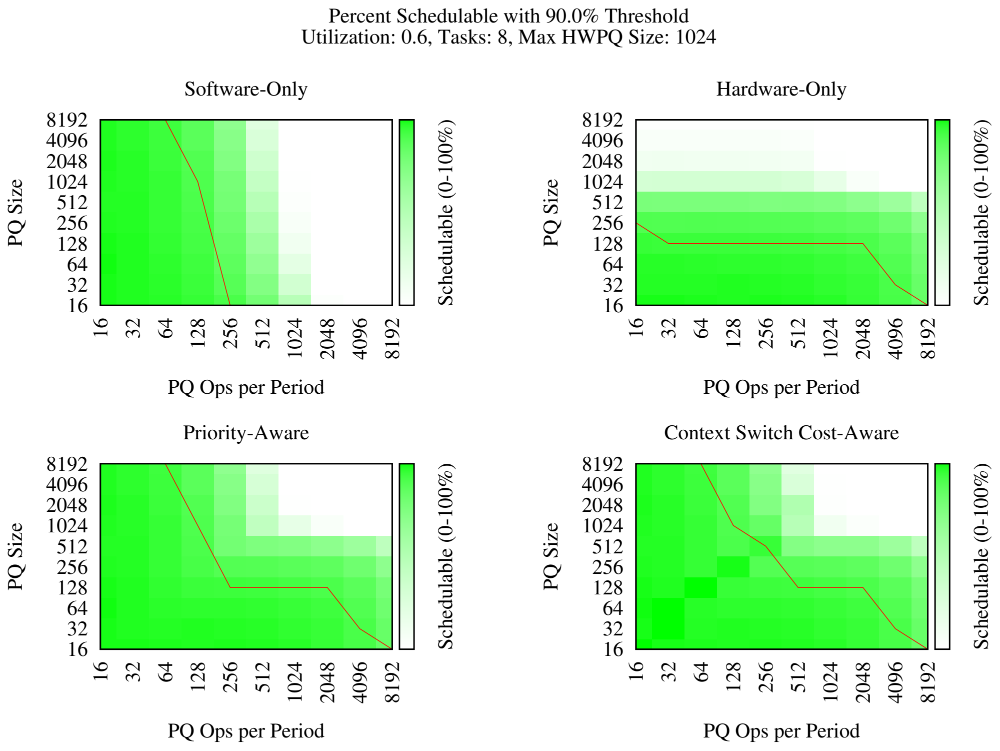

Figure 2: Schedulability of random task sets for utilization without PQ operations fixed at 0.6 and task set size at 8. Varying utilization and the number of tasks moves the threshold lines, which we show in later figures.

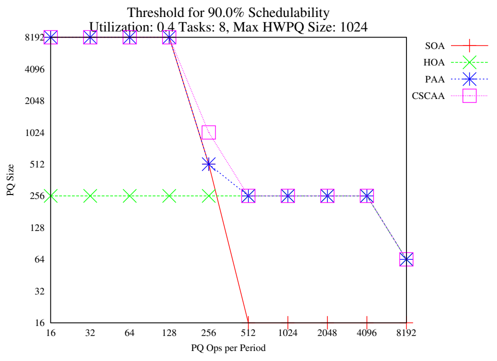

Figure 3: Schedulability with $U = 0.4$. As utilization decreases threshold lines move up because applications have greater spare utilization for larger PQs and more PQ operations.

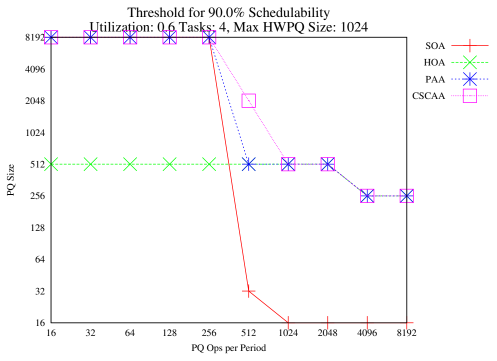

Figure 5: Schedulability with 4 tasks. As the number of tasks decreases the threshold lines move up. Halving the number of tasks more than doubles the number of schedulable task sets.

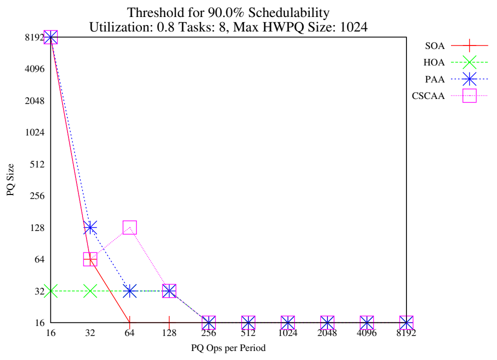

Figure 4: Schedulability with $U = 0.8$. As utilization increases threshold lines move down because applications cannot accommodate extra work induced by PQ operations.

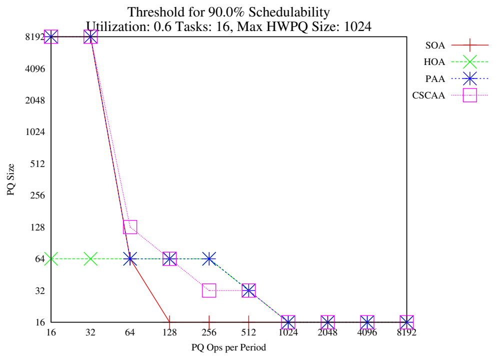

Figure 6: Schedulability with 16 tasks. As the number of tasks increases the threshold lines move down. Doubling the number of tasks reduces the number of schedulable task sets by more than half.

CSCAA with neither outperforming the other for all parameters although CSCAA generally does better than PAA.

Figure 3 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 4 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 PQ 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 PQ operations the task sets having PQ sizes twice as large are schedulable over 90% of the time with the extra 20% available CPU time.

Figure 5 again shows the threshold lines, this time with U at 0.6 and with 4 tasks; Figure 6 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 PQ operations (since every task does the same PQ workload). Doubling the tasks has the effect of reducing by a factor of two the PQ sizes of tasks sets that are schedulable at least 90% of the time for a given number of PQ operations (two factors if compared to half as many tasks and 20% more CPU time).

5.2 Real-world Applications

The synthetic task sets demonstrate HWPQs with the PAA and CSCAA algorithms can decrease utilization hence increase schedulability of applications that use priority queues. In this section we consider how HWPQs 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 PQ as a central data structure in their main algorithms. From the navigation domain we use a version of Dijkstra's algorithm that is executed on real-world maps taken from the DIMACS shortest path implementation challenge benchmarks [2]. From the terrain mapping domain we have 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 PQs [16]. We call these applications GPS and GWDT respectively. Both applications and their inputs are available online, see [16, 2].

In order to simulate these real-world applications we measured their behavior with respect to PQ parameters that affect HWPQ WCET; table 1 summarizes the measurements. We instrumented these applications with additional performance counters in order to measure the maximum PQ size, number (and type) of PQ operations, and the time taken by the PQ operations. For the GWDT application we included the peek, enqueue, and dequeue operations and also PQ allocation and freeing; the software PQ we used for the measurements was the 4-heap [16]. For the GPS application we included only enqueue and dequeue operations.

We executed these applications without modifications using timing mechanisms that are provided with the applications. These

App.	Input	PQ Size	PQ Operations	PQ 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%

Table 1: PQ behavior in real-world applications

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 we added timer calls before and after each PQ 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 we deducted from the sum of the time taken for PQ operations (because the time accounted toward the PQ operations includes half of the timer overhead). Then the ratio of the time taken for PQ operations to the total time taken of the unmodified application is a measure for the amount of time spent by the application in the PQ.

Using the parameters that we measured from running the applications we model 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 we generate 10000 random task sets 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 PQ parameters we obtained from measuring the applications. In particular we determine the WCET of a software PQ (using our numbers from the implicit binary heap) for the maximum PQ size and number of PQ operations for the task and use the percent of time the task should spend on the PQ to determine how long its total compute time should be. Then we compute the task's period by dividing its total compute time by its randomly generated utilization. We regenerate any task set that does not pass the response time analysis.

The result of task set generation is a set of tasks that use a software PQ and whose task set has a utilization equal to a known value U. We then remove the software PQ WCETs 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 we use the amount the assignment algorithm improves (reduces) task set utilization.

Figure 7 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 dif-

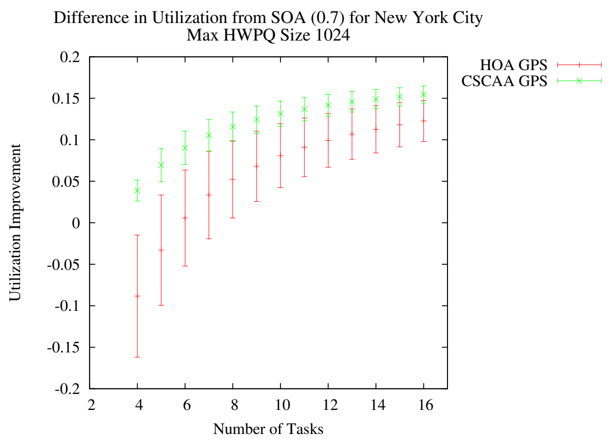

Figure 7: Utilization improvements for GPS application with increasing number of tasks executing local search in New York City.

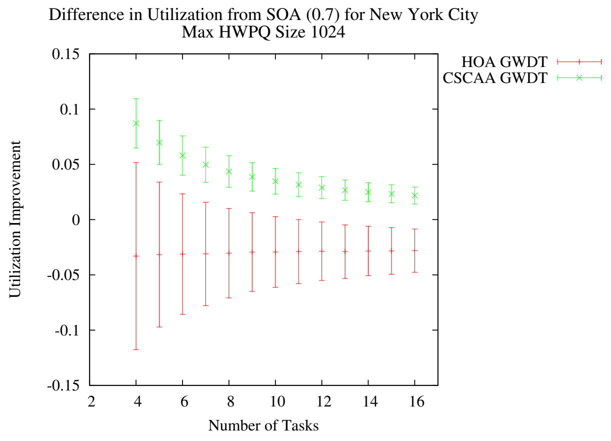

Figure 8: Utilization improvements for grey-weighted distance transform application with increasing numbers of tasks executing small GWDT.

ference 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. Larger numbers are better and represent the amount by which the utilization of the task set as a whole goes down; negative numbers indicate that the assignment algorithm does worse than SOA. Figure 8 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 HWPQ (HOA) yields rather 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 HWPQs 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 assignments of the HWPQ to tasks in these real-world task sets that improve task set utilization and therefore provide extra slack time in the system for other tasks to complete. The extra utilization can be useful for executing sporadic or background tasks.

6. RELATED WORK

This work builds on research in hardware queues, primarily of the FIFO and PQ varieties. HWPQs have been cited widely for both network routing and real-time scheduling. Moon et al. [19] compare four approaches to hardware PQs for high-speed networks and introduce an approach that melds two of the previous solutions. Kim and Shin [10] 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 [4] introduce a heap-based hardware PQ with pipelined stages of the enqueue and dequeue operations. The Spring Scheduling Coprocessor (SSCoP) [6] 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 HWPQ [22, 12, 11, 5, 13]. In contrast to the prior work, which focuses on hardware support for a single fixed-size PQ, our work demonstrates how arbitrarily-large PQs can share a HWPQ.

Carbon by Kumar et. al [14] provides hardware acceleration for multicore task scheduling with task LIFOs, prefetchers, and work stealing in hardware to support fine-grained thread-level parallelism. Carbon exposes a task queue API in the form of ISA extensions, so it is similar to the HWDS paradigm. It differs in that 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. Otherwise, the two approaches are not directly comparable.

Chandra and Sinnen [7] investigate HWDSs in the context of integrating Java with reconfigurable computing. The authors use a shift-register PQ to speed up Prim's minimum spanning tree algorithm. Their work uses a single PQ and HWPQ. In addition to the usual PQ operations, the authors investigate how to increase the queue length, use non-integer priority values, and add new operations. Our work differs from theirs by supporting large queues with an exception model instead of relying on library interpositioning on PQ accesses; we also allow multiple PQs to share the HWPQ.

For real-time systems a promising approach for providing a time-predictable memory system is to use a scratchpad memory [21]. Scratchpads can provide predictable access times and software control over code [29] and data [25]. Some problems with scratchpads include its limited size and the difficulty in choosing which data to store. The scratchpad memory management unit (SMMU) [27] uses custom hardware to split a virtual address space between a scratchpad and traditional RAM. SMMU uses runtime mechanisms to copy objects between the two memories so that once an object is moved into the scratchpad that object is accessed with predictable timing. The SMMU provides time predictability for accesses to the scratchpad and achieves WCET approaching that of the average-case performance with caching [26]. A downside for the SMMU is that it can actually increase WCET when applications exhibit poor temporal locality for object references because the time taken to copy an object into the scratchpad may negate any gain due to object reuse. Large objects also present a problem due to the spatial constraints of the scratchpad. HWDSs differ from scratchpads in that a HWDS relies on a high-level abstraction—the data structure—and provides support for high-level operations. Scratchpads rely on memory regions that contain an active working set without using any knowledge about application (data) behavior. So scratchpads may be generally more useful, yet HWDS have more knowledge available and are able to make use of known properties of data structures. HWDS uses these known properties to exploit parallelism and achieve speedup that scratchpads alone cannot. Combining the two approaches to use a HWDS with a scratchpad as the backing store—somewhat like how the SMMU splits storage between a scratchpad and RAM—may be an interesting research direction for reducing the WCET of overflow and underflow exception handlers.

7. CONCLUSION

In this paper we have demonstrated that HWDSs can benefit real-time systems by reducing worst-case execution times 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. We devised two new algorithms that assign tasks to use either a HWDS or a software DS and show those algorithms outperform just using the software DS or just using the HWDS for much of the explored application and parameter space. We demonstrated a HWPQ as an example of a HWDS and showed how real-world applications for navigation and image processing could obtain practical improvements in the range of 5–15% of total utilization when using our approach. Our results show promise for the HWDS approach and open new avenues of research into new HWDSs, improvements in the hardware to ease the integration of hardware and software, and better DS abstractions for programming.

Acknowledgments

This work is supported in part by NSF grant CNS-1117243, NSF grant CNS-0934725, AFOSR grant FA9550-09-1-0194, and George Washington University Summer Disssertation Fellowship. The authors thank the anonymous reviewers for their helpful advice.

8. REFERENCES

[1] RTEMS: Real-Time executive for multiprocessor systems. http://www.rtems.com/, 2011.

[2] 9th DIMACS implementation challenge: Shortest paths. http://www.dis.uniroma1.it/challenge9/download.shtml, 2012.

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

[4] 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.

[5] G. Bloom, G. Parmer, B. Narahari, and R. Simha. Real-Time scheduling with hardware data structures. In *Work-in-Progress Session. IEEE Real-Time Systems Symposium*, Dec. 2010.

[6] W. Burleson, J. Ko, D. Niehaus, K. Ramamritham, J. A. Stankovic, G. Wallace, and C. Weems. The spring scheduling

coprocessor: a scheduling accelerator. *IEEE Trans. Very Large Scale Integr. Syst.*, 7(1):38–47, 1999.

[7] R. Chandra and O. Sinnen. Improving application performance with hardware data structures. Tech. Report Faculty of Engineering, no. 678, University of Auckland, May 2010.

[8] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for Real-Time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '98, pages 16–30, London, UK, UK, 1998. Springer-Verlag.

[9] D. W. Jones. An empirical comparison of priority-queue and event-set implementations. *Commun. ACM*, 29(4):300–311, Apr. 1986. ACM ID: 5686.

[10] B. K. Kim and K. 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.

[11] P. Kohout, B. Ganesh, and B. 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.

[12] P. Kuacharoen, M. A. Shalan, and V. J. M. 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.

[13] C. Kumar, S. Vyas, J. Shidal, R. Cytron, C. Gill, J. Zambreno, and P. Jones. Improving system predictability and performance via hardware accelerated data structures. In *Dynamic Data Driven Application Systems (DDDAS)*, 2012.

[14] S. Kumar, C. J. Hughes, and A. 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.

[15] A. LaMarca and R. Ladner. The influence of caches on the performance of heaps. *J. Exp. Algorithmics*, 1, Jan. 1996. ACM ID: 235145.

[16] C. L. Luengo Hendriks. Revisiting priority queues for image analysis. *Pattern Recogn.*, 43(9):3003–3012, Sept. 2010. ACM ID: 1808374.

[17] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33:92–99, Nov. 2005. ACM ID: 1105747.

[18] N. Mhatre. *A Comparative Performance Analysis of Real-Time Priority Queues*. Master's thesis, The Florida State University, 2001.

[19] S. Moon, K. 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.

[20] F. Mueller. Timing analysis for instruction caches. *Real-Time Syst.*, 18(2/3):217–247, May 2000.

[21] I. Puaut and C. 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, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.

[22] 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.

[23] D. Sanchez, R. M. Yoo, and C. 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.

[24] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 154–165, 2003.

[25] Q. Wan, H. Wu, and J. 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, pages 41–50, New York, NY, USA, 2012. ACM.

[26] 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.

[27] 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, Apr. 2010.

[28] M. V. Wilkes. The memory gap and the future of high performance memories. *SIGARCH Comput. Archit. News*, 29(1):2–7, Mar. 2001. ACM ID: 373576.

[29] H. Wu, J. Xue, and S. Parameswaran. Optimal WCET-aware code selection for scratchpad memory. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, pages 59–68, New York, NY, USA, 2010. ACM.

[30] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23:20–24, Mar. 1995. ACM ID: 216588.