Cost-effective Slack Allocation for Lifetime Improvement in NoC-based MPSoCs

Brett H. Meyer, Adam S. Hartman, and Donald E. Thomas
Carnegie Mellon University
Department of Electrical and Computer Engineering
Pittsburgh, PA 15213
{bhm, ahartman, thomas}@ece.cmu.edu

Abstract—Wear-out related permanent faults are projected to make system lifetime a critical issue for all designs. In embedded systems, lifetime can be increased using slack, underutilization in execution and storage resources, so that when components fail, data and tasks can be re-mapped and re-scheduled. The design space of possible slack allocation is both large and complex. However, based on the observation that useful slack is often quantized, we have developed an approach that effectively and efficiently allocates execution and storage slack to jointly optimize system lifetime and cost. While exploring less than 1.4% of the slack allocation design space, our approach consistently outperforms alternative slack allocation techniques to find sets of designs within 1.4% of the lifetime-cost Pareto-optimal front.

I. INTRODUCTION
Recent research has shown that when integrated circuit wear-out failure mechanisms are not compensated for, system failure rates increase 365% when scaling from 180 to 65 nm, and exponentially beyond [1]. This lifetime degradation is in large part due to various parameters, including current density, $V_{Th}$, and $V_{DD}$, not scaling ideally as features shrink. This leads to higher system temperatures, worse sub-threshold leakage, and thus, faster device breakdown. As a result, all systems must be designed with lifetime in mind.

This paper is focused on design-time computer system architecture optimization for embedded network-on-chip multi-processor systems-on-chip (NoC-based MPSoCs). When a processor or switch fails in an NoC-based MPSoC, repair is impossible. However, the system may continue to operate if the design includes sufficient slack—execution and storage resources beyond those required in the initial configuration. We assume that systems can automatically detect component failure (for example, using [2]), at which point they reboot, re-mapping tasks and data from failed resources to those with slack and re-routing traffic. If performance constraints are still satisfied, system failure is averted [3].

Given a performance- and cost-constrained application and a fixed (NoC-based) communication architecture, our objective is to select where in the system to allocate slack, and how much to allocate, such that system lifetime and system manufacturing cost are jointly optimized. The design space of possible distributions of slack for a single communication architecture is large and complex: the number of possible slack allocations is exponential in the number of resources in the system and alternatives in the component library, and evaluating any single slack allocation requires repeated system-level performance, power, and temperature modeling.

To address the complexity of slack allocation, we have developed Critical Quantity Slack Allocation (CQSA), a novel, scalable, generalizable execution and storage slack allocation technique. CQSA takes advantage of the fact that the extra slack required to survive component failure is often quantized; we call these amounts of slack critical quantities. CQSA then uses the critical quantities for the network switches in the communication architecture of the NoC to efficiently focus the search for the best possible lifetime-cost trade-offs. Since the complexity of CQSA is closely tied to the number of switches in the NoC, the search grows much more slowly than the design space as systems grow.

Ours is the first work to: (1) define and use critical quantities of slack as the basis for a slack allocation technique; and, (2) allocate storage and execution slack to jointly optimize the cost and lifetime of MPSoCs, preventing system failure even when on-chip interconnect fails and memories are lost.

We show that CQSA outperforms a variety of other slack allocation approaches. On average, it searched less than 1.4% of the design space but was within 1.4% of the global lifetime-cost Pareto-optimal front found by exhaustive search.

II. RELATED WORK
Our work optimizes the system architecture at design-time for lifetime and cost, and is orthogonal to run-time lifetime management approaches such as RAMP [4], and other techniques that, for example, carefully manage the wear due to thermal cycling [5].

At the micro-architectural level, recent work has extended system lifetime by adding redundant functional units [6]. Another technique extends NoC lifetime by re-configuring the network when failures occur within switches [7]. Our work at the system level is orthogonal to these lower-level techniques.

At the system level, a large body of design-for-reliability research has improved upon older series-parallel systems and $n$-modular redundancy. COFTA uses error transparency to extend task-based fault tolerance techniques, inexpensively
enabling systems to both detect and operate in the presence of failure [8]. While this approach determines at design-time which resources are to be used when a component fails, our approach utilizes any component with slack to accommodate the failure of any other component of the same type.

Two closely related approaches perform slack allocation to jointly optimize the cost and lifetime of MPSoCs. The first begins by selecting processing elements to minimize area, and then makes incremental changes to processor selection to iteratively improve system lifetime [3]. The second approach performs joint performance-cost-lifetime optimization [9]. Our work extends these by considering storage slack and using critical quantities to organize an efficient search for Pareto-optimal lifetime-cost trade-offs.

III. SYSTEM-LEVEL LIFETIME OPTIMIZATION

At the system level, the lifetime of NoC-based MPSoCs can be improved at design-time by: (1) allocating execution slack by replacing low-performance processors with higher-performance processors, (2) allocating storage slack by replacing low-capacity memories with higher-capacity memories, or (3) changing the communication architecture by adding or modifying switches and links, and possibly adding spare processors and memories. The task of system-level design-for-resilience is to determine how to use these strategies to cost-effectively extend lifetime.

Slack allocation on a fixed communication architecture (1 and 2) is an extraordinarily complex problem. Given a communication architecture, the slack allocation design space contains up to $n^m$ designs for a system with $n$ components and $m$ possible alternatives in the component library: an MPEG-4 decoder example with 21 processors and 5 memories has 1.6 billion possible slack allocations. Since slack allocation is so complex and furthermore is needed to perform communication architecture exploration (3), we have focused on slack allocation. Jointly exploring slack allocation and communication architecture design is the subject of future work.

IV. CRITICAL QUANTITY SLACK ALLOCATION

To address the complexity of slack allocation, we have developed Critical Quantity Slack Allocation (CQSA), a novel, scalable execution and storage slack allocation technique. Given: (1) a description of a performance-constrained application, including computation, storage and communication requirements for each task; (2) a fixed communication architecture for a single-chip multiprocessor, including an initial selection of processors, memories, switches and their interconnection; and, (3) an initial task-resource mapping, including an assignment of computational tasks to processors, storage tasks to memories, and communication to links and switches; CQSA jointly optimizes system lifetime and cost by determining how much execution and storage slack should be allocated in the system and where in the system it should be allocated.

CQSA takes advantage of the fact that the extra slack required to survive component failure is often quantized. For a system to survive the failure of a particular processor, enough slack must be allocated so all of its tasks can be re-mapped. Allocating too much or too little slack serves no purpose, and actually degrades lifetime by unnecessarily increasing system power and therefore temperature.

We define the amount of slack which must be allocated to cover the functionality of a particular component as the critical quantity of slack for that component. A critical quantity for a component $C$ is denoted $(es, ss)$, and captures the execution slack $es$ (e.g., in MIPS) and storage slack $ss$ (e.g., in KB) necessary to replace the resources that would become inaccessible if $C$ were to fail. Processors generally have critical quantities of execution slack only (e.g., $(es, 0)$). Memories generally have critical quantities of storage slack only, (e.g., $(0, ss)$). Switches, however, which attach both processors and memories to the rest of the network, may have critical quantities of both execution and storage slack.

Switch critical quantities for an example architecture are illustrated in Figure 1. The task graph in Figure 1a is implemented in Figure 1b. All execution tasks (clouds) are assumed to be 250 MIPS; task EX is mapped to processor (square) PX. All storage tasks (cylinders) are assumed to be 1MB; task SX is mapped to memory (rounded rectangle) MX. We call Figure 1b the baseline architecture as it allocates no slack: each resource has been selected to match the requirements of the execution and storage tasks mapped to them. All of the processors are ARM9s (250 MIPS), and all of the memories are 1 MB SRAMs. This architecture has two unique switch critical quantities, illustrated with shaded gray boxes: $(750, 0)$ for Switch 3, and $(500, 1024)$ for Switches 1-2 and 4.

CQSA exploits the fact that system lifetime is most cost-effectively increased when slack is allocated such that system switch failure is survivable. Many slack allocations may address failures in a given set of resources, e.g., processors P1, P2 and memory M1 in Figure 1b. However, by allocating slack to cover failures in these components as well as the failure of Switch 1, lifetime is even further extended at no additional cost. By focusing search around the critical quantities of slack for switches, CQSA partitions the design space, and as a result, the complexity of CQSA grows slowly as the total number of components increases.
Algorithm 1 Critical Quantity Slack Allocation

1: // Stage 0
2: es = 0
3: while es < min(cqExecutionList) do
4:    sys = allocateExecSlackGreedily(sys)
5:    es = executionSlack(sys)
6: end while
7: // Stage 1
8: for all (es, 0) in cqExecutionList do
9:    sys = allocateExecSlackExhaustively(sys)
10:   while es ≤ es\_max do
11:      sys = allocateExecSlackGreedily(sys)
12:      es = executionSlack(sys)
13: end while
14: // Stage 2
15: for all (es, ss) in cqExecutionStorageList do
16:    sys = allocateExecStorageSlackExhaustively(sys, ss)
17:    while es ≤ es\_max and ss ≤ ss\_max do
18:      ssAllocation = allocateStorageSlackGreedily(sys)
19:      sys = compareTradeOff(esAllocation, ssAllocation)
20:      es = executionSlack(sys)
21:      ss = storageSlack(sys)
22: end while

A. Discussion of CQSA Stages

CQSA (Algorithm 1) is divided into three stages, each of which addresses different critical quantities, enabling it to discover different regions of the global Pareto-optimal set of slack allocations. CQSA begins with Stage 0, which addresses execution slack allocations not covered by a critical quantity. Critical quantities of execution slack alone ((es, 0), e.g., for processors, or switches attached to only processors) go in cqExecutionList and are processed in Stage 1. Critical quantities of execution and storage slack (including (0, ss), e.g., for memories, or switches attached to a combination of processors and memories) go in cqExecutionStorageList and are processed in Stage 2.

We will use Figure 1 for our running example. Execution slack is allocated by replacing ARM9s (250 MIPS) with ARM11s (500 MIPS). Storage slack is allocated by replacing 1MB SRAMs with 2MB SRAMs. The switch critical quantity (750, 0) (Switch 3) will be the basis for Stage 1, while (500, 1024) (Switches 1, 2 and 4) will be the basis for Stage 2. The lifetime (1/MTTF, 1/years) and cost (area, mm\(^2\)) of the baseline design is labeled A in Figure 2.

Stage 0 allocates slack to protect against the failure of single or small numbers of processors, potentially significantly improving lifetime at little cost. Starting from the baseline architecture, it finds the best low-cost allocations of execution slack. Stage 0 greedily allocates the minimum quanta of execution slack up to (but not including) the smallest execution-slack-only critical quantity in cqExecutionList (lines 2-5). 750 MIPS in our example. In our example, MTTF is first maximized when P9 is allocated slack (B in Figure 2), which leads to a 13% increase in lifetime relative to the baseline A. P9 is selected because the network around it is under the least bandwidth pressure, enabling a wide variety of task re-mappings and traffic re-routings. Lifetime is subsequently maximized when P5 is also allocated slack (C in Figure 2); this improves lifetime another 6% relative to the baseline A.

Stage 1 finishes the execution slack exploration started in Stage 0 by considering situations where careful execution slack allocation may result in the failure of a particular switch being survivable. Stage 1 is executed once for each critical quantity (es, 0), es > 0 in cqExecutionList, the list of execution-slack-only critical quantities. In our example, this list contains the critical quantity for Switch 3, (750, 0). First, an exhaustive search is performed to find the allocation of es MIPS (e.g., 750 MIPS) of execution slack that maximizes MTTF (line 9). In our example, Stage 1 evaluates each of \( \binom{9}{3} = 84 \) possible slack allocations before selecting design D in Figure 2: P5, P8 and P9 are made into ARM11s, and the failure of Switch 3 is now survivable. This improves lifetime by 25% compared with the baseline architecture. Stage 1 then proceeds to repeatedly greedily maximize MTTF by allocating the smallest quanta of execution slack to the exhaustive solution (lines 10-12). In our example, this produces a series of designs that can survive the failure of Switch 3, ending with design E (Figure 2), which improves lifetime up to 33% relative to the baseline A.

Stage 2 is the centerpiece of CQSA as it additionally considers storage slack. Starting from the baseline architecture once more, Stage 2 generates more points along the globally Pareto-optimal front by first performing an exhaustive search for the MTTF-optimal allocation of (es, ss), and then greedily allocating execution and storage slack. Stage 2 is executed once for each critical quantity (es, ss), es ≥ 0, ss > 0 in cqExecutionStorageList, the list of critical quantities of execution and storage slack. In our example, this list contains the critical quantity for Switches 1, 2 and 4, (500, 1024). First, an exhaustive search is used to find the allocation of es MIPS and ss KB that maximizes MTTF (line 16). For our example, this involves evaluating 108 different allocations. The lifetime and cost of the selected slack allocation is labeled F in Figure 2. This slack allocation improves lifetime by 24% with respect to the baseline A, and can survive the failure of either Switch 2 or Switch 3. If Switch 1 fails, 250 MIPS and 1024 KB of allocated slack is inaccessible, and the system fails. However, because this slack allocation increases cost by 25%
with respect to the baseline, it is not on the Pareto-optimal front. Stage 2 then begins comparing designs based on this initial slack allocation. Two new slack allocations are derived: one that incrementally increases execution slack (line 18), and another that incrementally increases storage slack (line 19). Each of these slack allocations is found with a greedy search that maximizes MTTF. The allocation with the best cost-MTTF trade-off is selected (line 20), providing the new starting point for another iteration (lines 18-19). This process repeats until no additional slack allocation is possible.

In our example, this process eventually results in Pareto-optimal designs, beginning with the design labeled \( G \) in Figure 2, resulting in a lifetime improvement of 33% with respect to the baseline, at a 32% cost increase. Stage 2 continues to allocate execution and storage slack, eventually resulting in design \( H \), which, with respect to the baseline, improves lifetime by 50% and increases area by 62%.

### B. Termination, Complexity and Scaling

The Pareto-optimal slack allocations are shown in Figure 2. From the baseline \( A \) to point \( H \), lifetime varies 50% and cost varies 62%. With many interesting trade-offs in between, a designer can select the best.

Expressing the complexity of CQSA generally is very difficult, as it is dependent on the number of components in the system, the available component library, and the initial system architecture (including component selection). Therefore, detailed analysis of the complexity of CQSA is omitted due to space considerations. We observe, however, that the number of unique critical quantities grows no faster than the number of switches in the system, which itself grows no faster than the number of components in the system. Later we will see that even when the design space grows five orders of magnitude in our examples, the number of designs evaluated by CQSA grows by only an order of magnitude.

### V. Experimental Setup

We performed two sets of experiments to evaluate CQSA. First, we used smaller benchmarks where we could exhaustively search for the global Pareto-optimal set of designs. This allowed us to precisely quantify our results. Next, we used a much larger benchmark for which exhaustive search is intractable. This allowed us to investigate how well CQSA scales as design spaces scale.

#### A. Applications and Architectures

We used three different benchmarks in our experiments: Multi-window Display (MWD) [10], and two MPEG-4 decoders, Core Profile Level 1 (CPL1) and Core Profile Level 2 (CPL2) [11]. We determined application computation and storage needs (and thus critical quantities of slack) using profiling. We constructed system architectures for each application using components presented with Figure 1b.

Table 1 summarizes the characteristics of the applications and architectures, and the computational cost of evaluating them. The number of slack allocations (SA) ranges from 11,664 for MWD to over 1.6 billion for CPL2. The architectures range from 3-switch (3-s) through 10-switch designs. The measured minimum (Min) and maximum (Max) times required to evaluate a single slack allocation on a Pentium 4 workstation is shown in seconds. Total lists the minimum time required to exhaustively evaluate all slack allocations.

#### B. System Cost and Lifetime Modeling

System (manufacturing) cost is the area of the floorplanned system, determined using Parquet [12], assuming a 90 nm process. Processor area is based on data sheet values [13]. SRAM area is derived using CACTI 5.3 [14]. All network routers are equivalent to the Alpha 21364’s on-chip router; we used ORION [15] to derive their area.

System lifetime is estimated using Monte Carlo Simulation (MCS) to sample possible sequences of component failures and determine how long, on average, performance constraints are satisfiable. To model component failure, we adopted a lognormal failure distribution model for each of three temperature-dependent failure mechanisms [6]: electromigration, time-dependent dielectric breakdown, and thermal cycling. The MTTF of each failure mechanism is normalized 30 years for the characterization temperature of 345 K [16].

For each MCS sample, lognormal failure times are randomly selected from the failure distributions of each failure mechanism of each component. To determine these failure distributions, we first estimate the temperature of the component. Component utilization is used to determine power dissipation. Processor power is derived using data sheet values, memory power using CACTI, and switch power using ORION. Using the floorplan and per-component power dissipation, HotSpot [17] finds the steady-state temperature for each component.

Tasks, data and traffic are re-mapped as components fail. If this remapping satisfies system performance constraints, failure is averted, and component utilization and temperature are then re-calculated so that component failure times may be updated accordingly [16]. Otherwise, we record the time \( F_i \) at which sample system \( i \) fails, ultimately estimating system MTTF with the sample mean \( \overline{F}_n \).

#### C. Pareto-Front Comparison Techniques

To reduce measurement bias, we used two different accuracy measures to compare the global Pareto-optimal fronts. Average Distance from Reference Set (ADRS) [18] is a distance-based accuracy measure. Given a reference set \( R \) (determined by exhaustive search) and a solution set \( S \) (the designs determined by the heuristic approaches), ADRS measures the average

<table>
<thead>
<tr>
<th>App.</th>
<th>SA</th>
<th>Arch.</th>
<th>Min (s)</th>
<th>Max (s)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>MWD</td>
<td>1.17e4</td>
<td>3-s</td>
<td>11.8</td>
<td>13.1</td>
<td>38.2 hours</td>
</tr>
<tr>
<td>CPL1</td>
<td>1.43e5</td>
<td>4-s</td>
<td>20.4</td>
<td>23.1</td>
<td>33.7 days</td>
</tr>
<tr>
<td>CPL2</td>
<td>1.61e9</td>
<td>10-s</td>
<td>46.4</td>
<td>144.5</td>
<td>2372.4 years</td>
</tr>
</tbody>
</table>
normalized best-case distance along the worst-case axis (cost or lifetime) from each design in \( R \) to the nearest design in \( S \). Coverage Difference of Two Sets (D) [19] is a volume-based measure that compares the hyperareas dominated by two solution sets \( S \) and \( S' \) and bounded by the point \( O' \), \( O' \) being just outside maximum cost and maximum 1/MTTF points.

D. Alternative Slack Allocation Approaches

To evaluate CQSA, we compared its results with the exhaustively determined \( R \) and the Pareto-optimal front determined with three other slack allocation heuristics.

Optimal execution slack allocation (Optimal ESA) represents the upper bound on the lifetime-cost trade-offs that can be discovered by approaches that only consider execution slack (e.g., [3], [9]). It generates the set of Pareto-optimal designs that allocate only execution slack.

Greedy slack allocation (Greedy SA) is a simple greedy slack allocation approach that starts with the baseline design (no slack). Two new designs are then derived, one that adds execution slack and one that adds storage slack. The best cost-lifetime trade-off is selected and becomes the baseline from which two more designs are derived. This process continues until no more slack can be allocated.

Random slack allocation (Random SA) selects a random subset of all possible slack allocations. The sample size is set to the number of designs explored by CQSA.

VI. RESULTS

In our first set of experiments, we performed exhaustive slack allocation\(^2\) to fully determine the slack allocation design space for each architecture for each of MWD and CPL1. The reference set \( R \) is selected from each of these. We then compared \( R \) with the locally lifetime-cost Pareto-optimal sets of designs, or solution set \( S \), from CQSA, Optimal ESA, Greedy SA, and Random SA, summarized in Table II.

For each application (App) and architecture (Arch) combination, Table II lists the number of designs evaluated (No.) and corresponding performance (ADRS, and D(R,S)) of CQSA and the comparison heuristics. For CQSA, the number of critical quantities considered is also listed (CQ). For Random SA, ADRS and D are estimated as the sample means \( \overline{ADRS}_{n} \) and \( \overline{T_	ext{h}}_n, n = 1000 \), resulting in a 95% confidence interval of less than 1% of the sample mean in each case.

The Pareto-optimal fronts for CPL1 5-s (excluding Random SA) are plotted in Figure 3. The global Pareto-optimal reference set (Ref) is plotted as black diamonds, CQSA as orange triangles, Optimal ESA (Opt. ESA) as red squares, and Greedy SA as blue x’s. System area varies 74%, and system lifetime varies 22%, both with respect to the baseline \( A \).

We observe in Table II that in the case of CPL1 5-s, CQSA finds a set of designs within 2.42% of the optimal by average distance and 5.72% by coverage difference while only exploring 536 designs, or 0.38% of the design space (from Table I). Greedy SA, on the other hand, finds designs five times further away from the optimal compared with CQSA. Its Pareto-optimal front is 11.35% and 27.91% off optimal with respect to average distance and coverage difference, while exploring 137 designs. Greedy SA allocates storage slack early, and, unable to re-start like CQSA, diverges from the optimal around design \( B \). Optimal ESA does even worse, finding designs 21.39% and 36.57% off optimal with respect to average distance and coverage difference.

The communication architecture for CPL1 5-s is illustrated in Figure 4. The gray-shaded areas capture the components responsible for determining each switch’s critical quantity of slack. In this design, any single switch failure is survivable given enough slack, and four of the five switches are connected to memories. As a result, storage slack is present in the overwhelming majority of Pareto-optimal designs. Execution slack alone is able to improve lifetime up to 11% for a 15% increase in area (design \( A \) to \( C \)). The inclusion of storage slack doubles the lifetime improvement of execution slack alone, increasing lifetime by 22% (design \( C \) to \( D \)). CQSA makes it possible to efficiently find these trade-offs by (1) considering storage slack as well as execution slack and (2) focusing on the critical quantities of slack in the system.

We observe that CQSA is consistently the most accurate approach, on average finding designs within 0.95% of the optimal by average distance and 1.81% by coverage difference, while exploring 1.7% of the design space. No other approach consistently found near optimal designs (e.g., both Optimal ESA and Greedy SA have errors of over 10% for CPL1 5-s).

\(^2\)We used Condor (http://www.cs.wisc.edu/condor) to evaluate large batches of designs in parallel, making exhaustive search tractable for MWD and CPL1.
In the second set of experiments we evaluated CQSA’s scalability by finding Pareto-optimal designs for the larger CPL2 benchmark. The design space for CPL2 is five orders of magnitude larger than that of MWD: it is too large to explore exhaustively or consider Random SA. Instead, we compared CQSA with Greedy SA and Optimal ESA directly using D.

**Observed Ref** represents the observed global Pareto-optimal front based on extensive low-sack exploration and all other data points explored by either Greedy SA or any run of CQSA, more than 205K of the 1.6 billion possible slack allocations. In this case, Optimal ESA is also approximated based on extensive low-sack exploration, in which we evaluated nearly 32K of the 10 million possible slack allocations. Table III summarizes the results of this experimentation. For each comparison approach, the table lists the number of designs evaluated (No.), the fraction of hyperarea dominated by CQSA but not the comparison approach \((D(CQSA, S), S)\), and the fraction of the hyperarea dominated by the comparison approach but not by CQSA \((D(S, CQSA))\).

For CPL2, CQSA uses six critical quantities, evaluating 6066 designs. We observe in Table III that CQSA is again the most accurate approach. 10.6% of the hyperarea it dominates is uncovered by Optimal ESA, while 4.09% is not dominated by Greedy SA. Of the hyperareas dominated by Greedy SA and Optimal ESA, CQSA dominates all but 0.27% and 0.03% respectively. We further observe that despite the fact that the CPL2 design space is 10^5 times larger than that for MWD, the number of designs evaluated by CQSA grows by only 10x. This slow rate of growth in exploration complexity is due to the fact that the number of evaluated critical quantities grew from four (for MWD 4-s) to six (for CPL2).

**VII. CONCLUSIONS**

We have defined and demonstrated a slack allocation technique, CQSA, that uses the critical quantities of slack for switches in NoC-based MPSoCs to efficiently search for the best lifetime-cost trade-offs. For smaller examples, CQSA found designs within 1.4% of the lifetime-cost Pareto-optimal front determined by exhaustive search while exploring on average only 1.4% of the design space. Execution slack alone produced sets of designs up to 21% worse than the global Pareto-optimal. For a larger example, CQSA performed the best, finding a Pareto-optimal front covering 10% more design space than an approach considering execution slack alone.

Since the complexity of CQSA is related to the number of switches in the NoC, its complexity grows sub-linearly with the number of components in the system. In our examples, even as the design space grew five orders of magnitude, the execution time only increased one order of magnitude.

**ACKNOWLEDGEMENTS**

This work was supported by the Semiconductor Research Corporation through contract 2008-HJ-1795.

**REFERENCES**


