Mutable Protection Domains: Towards a Component-based System for Dependable and Predictable Computing [*]

Gabriel Parmer and Richard West

Computer Science Department
Boston University
Boston, MA 02215
{gabep1, richwest}@cs.bu.edu

Abstract

The increasing complexity of software poses significant challenges for real-time and embedded systems beyond those based purely on timeliness. With embedded systems and applications running on everything from mobile phones, PDAs, to automobiles, aircraft and beyond, an emerging challenge is to ensure both the functional and timing correctness of complex software. We argue that static analysis of software is insufficient to verify the safety of all possible control flow interactions. Likewise, a static system structure upon which software can be isolated in separate protection domains, thereby defining immutable boundaries between system and application-level code, is too inflexible to the challenges faced by real-time applications with explicit timing requirements. This paper, therefore, investigates a concept called "mutable protection domains" that supports the notion of hardware-adaptable isolation boundaries between software components. In this way, a system can be dynamically reconfigured to maximize software fault isolation, increasing dependability, while guaranteeing various tasks are executed according to specific time constraints. Using a series of simulations on multi-dimensional, multiple-choice knapsack problems, we show how various heuristics compare in their ability to rapidly reorganize the fault isolation boundaries of a component-based system, to ensure resource constraints while simultaneously maximizing isolation benefit. Our `ssh oneshot` *algorithm offers a promising approach to address system dynamics, including changing component invocation patterns, changing execution times, and mispredictions in isolation costs due to factors such as caching.*

[*]This material is based upon work supported by the National Science Foundation under Grant Numbers 0615153 and 0720464. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

1 Introduction

As the complexity of emerging real-time and embedded software systems increases, new challenges beyond those focusing solely on timeliness guarantees are becoming increasingly significant. In particular, it is expected that embedded devices such as mobile phones and personal digital assistants will support tens of millions of lines of code in the foreseeable future. Web services, video-on-demand and multimedia databases are already appearing on hand-held devices and so it makes sense that software complexity will only increase over time. Consequently, it will be impossible to entirely verify the correctness of a large software system statically. For example, run-time interactions between various threads of execution, asynchronous events (e.g., interrupts from I/O devices), and various dynamically-generated references to memory locations make it impossible to guarantee the behavioral correctness of a body of software at compile-time. From a CPU protection perspective, determining whether a thread will terminate is, in general, undecidable. We are therefore faced with the challenge of limiting the scope of potentially misbehaving, or faulty, software on overall system functionality.

The above argues that static verification of software correctness is a daunting challenge, and in many cases insufficient for complex software systems. It makes sense to leverage, where appropriate, hardware and software-based fault isolation (i.e., protection) mechanisms [16, 4, 18, 8], to limit the scope of adverse side-effects caused by errant software. Fault isolation overheads (e.g., due to page-table management, or run-time software safety checks) impact the granularity at which they can be imposed. They in turn impact the predictability of software execution, because factors such as TLB/cache misses, page replacement policies, garbage collection, and memory-bounds checks impose variable costs.

Fault isolation provisions of modern systems are typically limited to coarse-grained entities, such as segments that separate user-space from kernel-level, and processes that encapsulate system and application functionality. For example, systems such as Linux simply separate user-space from a monolithic kernel address space. Micro-kernels provide finer-grained isolation between higher-level system services, often at the cost of increased communication overheads. Common to all these system designs is a static structure, that is inflexible to changes in the granularity at which fault isolation can be applied.

For the purposes of ensuring behavioral correctness of a complex software system, it is desirable to provide fault isolation techniques at the smallest granularity possible, while still ensuring predictable software execution. For example, while it may be desirable to assign the functional components of various system services to separate protection domains, the communication costs may be prohibitive in a real-time setting. That is, the costs of marshaling and unmarshaling message exchanges between component services, the scheduling and dispatching of separate address spaces and the impacts on cache hierarchies (amongst other overheads) may be unacceptable in situations where deadlines must be met. Conversely, multiple component services mapped to a single protection domain experience minimal communication overheads but lose the benefits of isolation from one another.

Given the above, this paper investigates the design of a system with "mutable protection domains" (MPDs), that is flexible in its placement of fault isolation boundaries around various application and system components. Where possible, we attempt to maximize fault isolation, by mapping fine-grained software components to separate hardware protection domains, at the expense of increased communication overheads. In situations where such fine-grained isolation violates the acceptable end-to-end communication costs through a series of component services that are required to meet specific deadlines, we strategically increase the isolation granularity. By allowing dynamic changes to system structure, we are able to consider diverse hardware capabilities of numerous embedded computing platforms upon which systems software is deployed. Our system design assumes that certain isolation boundaries have preference over others. Using a combination of isolation benefit values applied to the boundaries between components, and the communication costs between components, we show how to dynamically restructure a component-based system to maximize isolation utility while maintaining timeliness.

The rest of the paper is organized as follows. Section 2 provides an overview of the system under consideration, and formally defines the problem being addressed. System dynamics and proposed solutions to the problem are then described. Section 3 then briefly describes an implementation of a system with mutable protection domains. An experimental evaluation is covered in Section 4, followed by related work in Section 5. Finally, conclusions and future work are discussed in Section 6.

2 System Overview

With mutable protection domains, isolation between components is increased when there is a resource surplus, and is decreased when there is a resource shortage. Such a system, comprising fine-grained components or services, can be described by a directed acyclic graph (DAG), where each node in the graph is a component, and each edge represents inter-component communication (with the direction of the edge representing control flow). Represented in this fashion, a functional hierarchy becomes explicit in the system construction. Multiple application tasks can be represented as subsets of the system graph, that rely on lower-level components to manage system resources. Component-based systems enable system and application construction via composition and have many benefits, specifically to embedded systems. They allow application-specific system construction, encourage code reuse, and facilitate quick development. Tasks within the system are defined by execution paths through a set of components.

A natural challenge in component-based systems is to define where protection domains should be placed. Ideally, the system should maximize component isolation (thereby increasing system dependability in a beneficial manner) while meeting constraints on application tasks. Task constraints can vary from throughput goals to memory usage, to predictable execution within a worst-case execution time (WCET). A task's WCET is formulated assuming a minimal amount of fault isolation present within the system. A schedule is then constructed assuming this WCET, and the implicit requirement placed on the system is that the task must complete execution within its allotted CPU share. In most cases, the actual execution time of tasks is significantly lower than the pessimistic worst case. This surplus processing time within a task's CPU allocation can be used to increase the isolation between components (inter-component communication can use the surplus CPU time).

In general, task constraints on a system with mutable protection domains can be defined in terms of multiple different resources. We focus on timeliness constraints of tasks in this paper, and consider only a single resource (i.e., CPU time) for communication between, and execution of, components. For n tasks in a system, each task, τ_k, has a corresponding target resource requirement, RT_k, which is proportional to its worst-case execution time. The measured resource usage, RM_k, is the amount of τ_k's resource share utilized by its computation. Similarly, the resource surplus for τ_k is RS_k, where $RS_k = RT_k - RM_k$. For n tasks the resource surplus is represented as a vector, $\vec{RS} = \langle RS_1, \ldots, RS_n \rangle$.

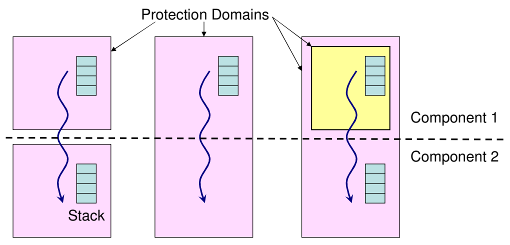


Figure 1. Example Isolation Levels.

In response to resource surpluses, different levels of isolation can be placed at the boundaries between components, depending upon their inter-component communication overheads. Three possible *isolation levels* are depicted in Figure 1. On the left, we see complete hardware isolation, equivalent to process-level protection in UNIX-type systems, which incurs costs in terms of context switching between protection domains. In the center, we have no isolation, equivalent to how libraries are linked into the address space of code that uses them. Such a lack of isolation implies only function-call overheads for inter-component communication. Finally, the right-hand side of Figure 1 depicts an asymmetric form of isolation, whereby component 1 is *inside* the protection domain of component 2 but not vice versa. This isolation scheme is equivalent to that found in many monolithic OSes such as Linux, which separate the kernel from user-space but not vice versa. It is also similar to the scheme used in our User-Level Sandboxing approach [20].

Problem Definition: By adapting isolation levels, which in turn affects inter-component communication costs, we attempt to increase the robustness of a software system while maintaining its timely execution. The problem, then, is to find a system configuration that maximizes the benefit of fault isolation, while respecting task execution (and, hence, resource) constraints. Using a DAG to represent component interactions, let $E = \{e_1, \ldots, e_m\}$ be the set of edges within the system, such that each edge, $e_i \in E$, defines an isolation boundary, or *instance*, between component pairs. For each edge, e_i, there are N_i possible isolation levels, where $N_{max} = max_{\forall e_i \in E}(N_i)$. Where isolation is required and immutable for security reasons, there might exist only one available isolation level ($N_i = 1$), so that security is never compromised. Isolation level j for isolation instance, e_i, is denoted e_{ij}. The overhead, or resource cost, of e_{ij} is $\vec{c_{ij}}$, where $c_{ijk} \in \vec{c_{ij}}, \forall RS_k \in \vec{RS}$. Conversely, each isolation level provides a certain benefit to the system, b_{ij}. We assume that the costs are always lower for lower isolation levels and that the benefit is always higher for higher isolation levels. Finally, $\vec{s}$ denotes a solution vector, where isolation level $s_i \in \{1, \cdots, N_i\}$ is chosen for isolation instance e_i. The solution vector defines a system isolation configuration (or system configuration for short).

More formally, the problem of finding an optimal system configuration is as follows:

$$\begin{aligned} &\text{maximize} \quad \sum_{i<m} b_{is_i} \\ &\text{subject to} \quad \sum_{i<m} c_{is_ik} \le RS_k, \ \forall RS_k \in \vec{RS} \\ &\qquad s_i \in \{1, \ldots, N_i\}, \ \forall e_i \in E \end{aligned} \tag{1}$$

Represented in this manner, we have a multi-dimensional, multiple-choice knapsack problem (MMKP). Though this problem is NP-Hard, approximations exist [9, 10, 2, 15]. Specifically heuristics proposed in these papers attempt to solve the objective function:

$$O(E, \vec{RS}) = \max_{0<j\le N_i} \{O(E \backslash e_i, \vec{RS} - \vec{c_{ij}}) + b_{ij} \,|\, e_i \in E\}$$

2.1 System Dynamics

Previous approaches to the MMKP for QoS attempt to solve a resource-constrained problem off-line, to maximize system utility. After a solution is found, actual system resources are then allocated. In our case, we wish to alter an existing system configuration *on-line*, in response to resource surpluses or deficits that change over time. The dynamics of the system that introduce changes in resource availability and isolation costs include: (i) threads changing their invocation patterns across specific isolation boundaries, thus changing the overhead of isolation instances throughout the system, (ii) threads altering their computation time within components, using more or less resources, thus changing $\vec{RS}$, and (iii) misprediction in the cost of isolation. Thus, heuristics to calculate system configurations to maximize beneficial isolation over time must adapt to such system dynamics.

It is difficult to compensate for these dynamic effects as the measurements that can be taken directly from the system do not yield complete information. Specifically, at each reconfiguration, the system can measure resource usage, $\vec{RM}$, but explicit information regarding the overhead of isolation is not in the general case observable. For example, the isolation costs between two components mapped to separate protection domains might include context-switch overheads, which in turn have secondary costs on caches, including translation look-aside buffers (TLBs). Such secondary costs are difficult to extract from the total runtime of a task. Section 2.6 discusses the impact of mispredicting isolation costs on system behavior.

2.2 Dynamic Programming Solution

Given that our problem can be defined as a multi-dimensional, multiple-choice knapsack problem, there are known optimal dynamic programming solutions, For comparison, we describe one such approach similar to that in [10].

$$DP[i,j,\vec{RS}] = \begin{cases} \max(-\infty, DP[i,j-1,\vec{RS}]) & if \ \forall_{RS_k \in \vec{RS}} c_{ijk} > RS_k \\ b_{ij} & if \ i = 1 \\ take(i,1,\vec{RS}) & if \ j = 1 \\ \max(take(i,j,\vec{RS}), DP[i,j-1,\vec{RS}]) & otherwise \end{cases}$$

$$take(i,j,\vec{RS}) \equiv b_{ij} + DP[i-1, N_{i-1}, \vec{RS} - \vec{c_{ij}}]$$

Figure 2. Dynamic programming solution.

Figure 2 shows the dynamic programming solution DP. The algorithm is initially invoked with $DP[|E|, N_{|E|}, \langle RS_1, \ldots, RS_n \rangle]$. For the lowest isolation level (level 1) of an edge, we assume the sum of the minimal isolation levels is always within the resource consumption limits. That is, $\forall k, \sum_{e_i \in E} c_{i1k} \leq RS_k$.

The recurrence keeps track of the current resource usage and iterates through all isolation levels for a given instance, choosing that which provides the best benefit given its resource cost. The base cases are when we have run out of resources or reached the last isolation instance ($i = 1$).

The complexity of the algorithm reflects the memory structure used: $O(|E| \cdot N_{max} \cdot RS_1 \cdot \ldots \cdot RS_n)$. Because of the memory and execution time requirements, this algorithm is impractical for on-line use, but is useful for comparison.

2.3 HEU

HEU is a heuristic solution first proposed in [9] that is summarized here. HEU is a popular comparison case in the MMKP literature and is competitive in terms of quality of solution. Previous algorithms, HEU included, assume that the knapsack is initially empty and choose items to place in it from there. This algorithm's general strategy is to weight isolation benefits versus their costs to choose which isolation level to increase. It uses Toyoda's notion of an aggregate resource cost [17] to collapse multiple constraint dimensions into one by penalizing those dimensions that are more scarce. Then, HEU uses a greedy algorithm based on benefit density to choose the isolation level that will yield the highest benefit for the least resource usage. This chosen isolation level is added to the system configuration. This process is repeated, new aggregate resource costs are chosen and that isolation level with the best benefit density is chosen until the resources are expended. Because the aggregate resource cost is recomputed or refined when choosing each edge, we will refer to this algorithm as using *fine-grained refinement*. The asymptotic time complexity of this algorithm is $O(|E|^2 \cdot N_{max}^2 \cdot |\vec{RS}|)$.

2.4 Computing Aggregate Resource Costs

Aggregate resource costs should have higher contributions from resources that are scarce, thereby factoring in the cost per unit resource. Inaccurate aggregate costs will lead to a system that does not evenly distribute resource usage across its task constraint dimensions. The approach we take to computing costs is similar to that in [10]. First, we compute an initial penalty vector which normalizes the total resource costs across all isolation instances and levels, $\vec{c_{ij}}$, by the vector of available resources, $\vec{RS}$. This is shown in Equation 2 and will be subsequently referred to as init_penalty_vect, $\vec{p}$.

$$\vec{p} = \langle p_1, \cdots, p_n \rangle \mid p_k \in \vec{p} = \frac{\sum_{\forall e_i \in E} \sum_{j \leq N_i} c_{ijk}}{RS_k} \qquad (2)$$

$$p_k = \frac{(\sum_{\forall e_i \in E} c_{is_ik}) p'_k}{(\sum_{\forall e_i \in E} c_{is_ik}) + RS_k} \mid p_k \in \vec{p}, p'_k \in \vec{p'} \qquad (3)$$

Equation 3 is used to refine the penalty vector, taking into account the success of the previous value, $\vec{p'}$. Recall from Section 2 that s_i is the chosen isolation level for isolation instance e_i, while c_{is_ik} is the cost in terms of resource constraint RS_k. We will subsequently refer to the updated penalty vector calculated from Equation 3 as update_penalty_vect. Finally, Equation 4 defines the aggregate resource cost, c^*_{ij}, using the most recent penalty vector, $\vec{p}$.

$$c^*_{ij} = \sum_{\forall RS_k \in \vec{RS}} (c_{ijk} - c_{is_ik})(p_k) \qquad (4)$$

2.5 Successive State Heuristic

Our approach to solving the multi-dimensional, multiple-choice knapsack problem differs from traditional approaches, in that we adapt a system configuration from the current state. By contrast, past approaches ignore the current state and recompute an entirely new configuration, starting with an empty knapsack. In effect, this is equivalent to solving our problem with a system initially in a state with minimal component isolation.

Our solution, which we term the *successive state heuristic* (ssh), successively mutates the current system configuration. ssh assumes that the aggregate resource cost, c^*_{ij}, for all isolation levels and all instances has already been computed, as in Section 2.4. Edges are initially divided into two sets: set H comprises edges with higher isolation levels than those in use for the corresponding isolation instances in the current system configuration, while set L comprises edges at correspondingly lower isolation levels. Specifically, $e_{ij} \in H$, $\forall j > s_i$ and $e_{ij} \in L$, $\forall j < s_i$. Each of the edges in these sets are sorted with respect to their change in *benefit density*, $(b_{ij} - b_{is_i})/c^*_{ij}$. Edges in L are sorted in increasing order with respect to their change in benefit density, while those in H are sorted in decreasing order. A summary of the `ssh` algorithm follows (see **Algorithm 1** for details):

(i) While there is a deficit of resources, edges are removed from the head of set L to replace the corresponding edges in the current configuration. The procedure stops when enough edges in L have been considered to account for the resource deficit.

(ii) While there is a surplus of resources, each edge in H is considered in turn as a replacement for the corresponding edge in the current configuration. If $e_{ij} \in H$ increases the system benefit density and does not yield a resource deficit, it replaces e_{is_i}, otherwise it is added to a dropped list, D. The procedure stops when an edge is reached that yields a resource deficit.

(iii) At this point, we have a valid system configuration, but it may be the case that some of the edges in H could lead to higher benefit if isolation were lessened elsewhere. Thus, the algorithm attempts to concurrently add edges from the remaining edges in both H and L. A new configuration is only accepted if it does not produce a resource deficit and heightens system benefit.

(iv) If there is a resource surplus, edges from the dropped list, D, are considered as replacements for the corresponding edges in the current configuration.

The cost of this algorithm is $O(|E| \cdot N_{max} \log(|E| \cdot N_{max}))$, which is bounded by the time to sort edges. The `ssh` algorithm itself is invoked via:

(1) **Algorithm 2**. Here, only an initial penalty vector based on Equation 2 is used to derive the aggregate resource cost, c^*_{ij}. The cost of computing the penalty vector and, hence, aggregate resource cost is captured within $O(|\vec{RS}| \cdot |E| \cdot N_{max})$. However, in most practical cases the edge sorting cost of the base `ssh` algorithm dominates the time complexity. We call this algorithm `ssh_oneshot` as the aggregate resource cost is computed only once.

(2) **Algorithm 3**. This is similar to Algorithm 2, but uses Equation 3 to continuously refine the aggregate resource cost given deficiencies in its previous value. The refinement in this algorithm is conducted after an entire configuration has been found, thus we say it uses *coarse-grained refinement*. This is in contrast to the fine-grained refinement in Section 2.3 that adjusts the aggregate resource cost after each *isolation level* is found. We found that refining the aggregate resource cost more than 10 times, rarely increased the benefit of the solution. This algorithm has the same time complexity as `ssh_oneshot`, but does add a larger constant overhead in practice.

Algorithm 1: `ssh`: Successive State Heuristic

```
Input: $\vec{s}$: current isolation levels, $\vec{RS}$: resource surplus
$b^*_{ij} = (b_{ij} - b_{is_i})/c^*_{ij}, \forall i, j$ // benefit density change
// sorted list of lower isolation levels
$L$ = sort_by_b*({$\forall e_{ij} | e_i \in E \wedge j < s_i$})
// sorted list of higher isolation levels
$H$ = sort_by_b*({$\forall e_{ij} | e_i \in E \wedge j > s_i$})
$D = \phi$          // dropped set (initially empty)
while $\exists k, RS_k < 0 \wedge L \neq \phi$ do       // lower isolation
    $e_{ij}$ = remove_head($L$)
    if $c^*_{ij} < c^*_{is_i}$ then
        $\vec{RS} = \vec{RS} + \vec{c_{is_i}} - \vec{c_{ij}}$
        $s_i = j$
end
$e_{ij}$ = remove_head($H$)
while $(\nexists k, RS_k + c_{is_ik} - c_{ijk} < 0 \vee b^*_{ij} \leq b^*_{is_i}) \wedge e_{ij}$ do
// raise isolation greedily
    if $b^*_{ij} > b^*_{is_i}$ then               // improve benefit?
        $\vec{RS} = \vec{RS} + \vec{c_{is_i}} - \vec{c_{ij}}$
        $s_i = j$
    else $D = D \cup e_{ij}$
    $e_{ij}$ = remove_head($H$)
end
replace_head($e_{ij}, H$)
// refine isolation considering both lists
$\vec{s'} = \vec{s}$
while $H \neq \phi \wedge L \neq \phi$ do
    repeat                              // expend resources
        $e_{ij}$ = remove_head($H$)
        if $b^*_{ij} > b^*_{is'_i}$ then             // improve benefit?
            $\vec{RS} = \vec{RS} + \vec{c_{is'_i}} - \vec{c_{ij}}$
            $s'_i = j$
        else $D = D \cup e_{ij}$
    until $\exists k, RS_k < 0 \vee H \neq \phi$
    while $\exists k, RS_k < 0 \vee L \neq \phi$ do   // lower isolation
        $e_{ij}$ = remove_head($L$)
        if $c^*_{ij} < c^*_{is'_i}$ then
            $\vec{RS} = \vec{RS} + \vec{c_{is'_i}} - \vec{c_{ij}}$
            $s'_i = j$
    end
    // found a solution with higher benefit?
    if $\sum_{\forall i} b_{is'_i} > \sum_{\forall i} b_{is_i} \wedge \nexists k, R_k < 0$ then $\vec{s} = \vec{s'}$
end
while $D \neq \phi$ do     // add dropped isolation levels
    $e_{ij}$ = remove_head($D$)
    if $j > s_i \wedge \nexists k, RS_k + c_{is_ik} - c_{ijk} < 0$ then
        $\vec{RS} = \vec{RS} + \vec{c_{is_i}} - \vec{c_{ij}}$
        $s_i = j$
end
return $\vec{s}$
```

2.6 Misprediction of Isolation Overheads

The proposed system can measure the number of component invocations across specific isolation boundaries to estimate communication costs. However, it is difficult to measure the cost of a single invocation. This can be due to many factors including secondary costs of cache misses,

Algorithm 2: `ssh_oneshot`

```
  Input: $\vec{RS}$: resource surplus, $\vec{s}$ solution vector
1 $\vec{p}$ = init_penalty_vect($\vec{RS}$, $\vec{s}$)
2 $c^*_{ij}$ = aggregate_resource($\vec{p}$, $\vec{c}_{ij}$), $\forall i, j$
3 $\vec{s}$ = ssh($\vec{s}$, $\vec{RS}$)
4 return $\vec{s}$
```

Algorithm 3: `ssh_coarse`

```
   Input: $\vec{RS}$: resource surplus, $\vec{s}$ solution vector
1  $\vec{p}$ = init_penalty_vect($\vec{RS}$, $\vec{s}$)
2  $i = 0$
   // refine penalty vector
3  while $i < 10$ do
4      $c^*_{ij}$ = aggregate_resource($\vec{p}$, $\vec{c}_{ij}$), $\forall i, j$
5      $\vec{s'}$ = ssh($\vec{s}$, $\vec{RS}$)
6      $\vec{p}$ = update_penalty_vect($\vec{RS}$, $\vec{p}$, $\vec{s'}$)
7      if $\sum_{\forall i} b_{is'_i} > \sum_{\forall i} b_{is_i}$ then
8          $\vec{s} = \vec{s'}$              // found better solution
9      $i$++
10 end
11 return $\vec{s}$
```

which can be significant [18]. Given that the cost of a single invocation can be mispredicted, it is essential to guarantee such errors do not prevent the system converging on a target resource usage. We assume that the average estimate of isolation costs for each resource constraint, or task, k, across all edges has an error factor of x_k, i.e., $estimate = x_k * actual\_overhead$. Values of $x_k < 1$ lead to heuristics underestimating the isolation overhead, while values of $x_k > 1$ lead to an overestimation of overheads. Consequently, for successive invocations of MMKP algorithms, the resource surplus is mis-factored into the adjustment of resource usage. As an algorithm tries to use all surplus resources to converge upon a target resource value, the measured resource usage at successive steps in time, $RM_k(t)$, will in turn miss the target by a function of x_k. Equation 5 defines the recurrence relationship between successive adjustments to the measured resource usage, $RM_k(t)$, at time steps, $t = 0, 1, ...$ When $x_k > 0.5$ for the misprediction factor, the system converges to the target resource usage, RT_k. This recurrence relationship applies to heuristics such as `ssh` that adjust resource usage from the current system configuration.

$$\begin{aligned}
RM_k(0) &= \text{resource consumption at } t = 0 \\
RM_k(t+1) &= RM_k(t) + x_k^{-1} RS_k(t) \mid RS_k(t) = RT_k - RM_k(t) \\
&= x_k^{-1} RT_k + (1 - x_k^{-1}) RM_k(t) \\
\\
RM_k(t) &= x_k^{-1} RT_k (\textstyle\sum_{i=0}^{t-1} (1 - x_k^{-1})^i) + (1 - x_k^{-1})^t RM_k(0) \\
RM_k(\infty) = &\begin{cases} RT_k & \text{if } x_k > 0.5 \\ \infty & \text{otherwise} \end{cases}
\end{aligned} \tag{5}$$

For algorithms that do not adapt the current system configuration, they must first calculate an initial resource usage in which there are no isolation costs between components. However, at the time such algorithms are invoked they may only have available information about the resource usage for the current system configuration (i.e., $RM_k(t)$). Using $RM_k(t)$, the resource usage for a configuration with zero isolation costs between components (call it RU_k) must be estimated. RU_k simply represents the resource cost of threads executing within components. Equation 6 defines the recurrence relationship between successive adjustments to the measured resource usage, given the need to estimate RU_k. In the equation, $\alpha_k(t)$ represents an estimate of RU_k, which is derived from the measured resource usage in the current configuration, $RM_k(t)$, and an estimate of the total isolation costs at time t (i.e., $x_k(\sum_{\forall i} c_{is_i}) \mid RM_k(t) - RU_k = \sum_{\forall i} c_{is_i}$).

$$\begin{aligned}
RM_k(0) = &\quad \text{resource consumption at } t = 0 \\
\alpha_k(t) = &\quad RM_k(t) - x_k(RM_k(t) - RU_k) \\
RM_k(t+1) = &\quad RU_k + x_k^{-1}(RT_k - \alpha_k(t)) \\
= &\quad x_k^{-1} RT_k + (1 - x_k^{-1}) RM_k(t)
\end{aligned} \tag{6}$$

Given that Equation 6 and 5 reduce to the same solution, heuristics that reconfigure a system based on the current configuration and those that start with no component isolation both converge on a solution when $x_k > 0.5$. Equation 7 allows the system to estimate the misprediction factor for total isolation costs. This equation assumes that overheads unrelated to isolation hold constant in the system.

$$\begin{aligned}
x_k &= \tfrac{RM_k(n-1) - RT_k}{RM_k(n-1) - RM_k(n)} \\
&= \tfrac{RS_k(n-1)}{RS_k(n-1) - RS_k(n)}
\end{aligned} \tag{7}$$

3 MPD System Implementation

In this paper, we focus on the resource management and algorithmic aspects of mutable protection domains (MPDs). In this section we briefly outline key design elements of the component-based system we are building to support MPDs. We have added the ability to interpose component services on the service requests of other components using a version of Hijack [14]. An overview of the architecture can be seen in Figure 3.

The challenge is supporting both direct functional invocation with little overhead when no isolation is present, and inter-component communication with argument marshaling though the kernel when isolation is required. To this end, every function that can be invoked across a component boundary is paired with a capability and its user-level shadow (`KCap` and `UCap` in the figure). The capability exists within the trusted kernel and bestows permission

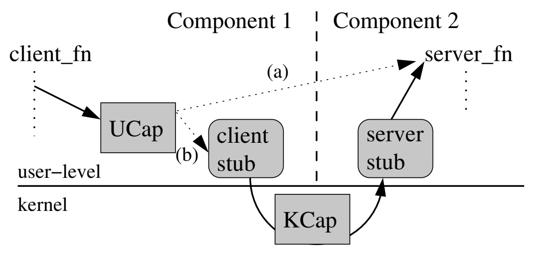


Figure 3. MPD Inter-component invocation.

to make a functional invocation across a component boundary. The capability contains information regarding the currently configured isolation level between the components. The shadow of the capability at user-level exists so that current configuration information can be easily accessed from user-level.

Specifically, the method by which a call from `client_fn` in component 1 to `server_fn` in component 2 depends on the isolation level between the two components. Functionality linked into component 1 examines `UCap` directly. Specifically, a function address is retrieved from the capability and invoked. In the case where no isolation is present, this invocation calls `server_fn` directly as in `(a)`. The `server_fn` returns directly to where it was invoked in `client_fn`. In the case that fault isolation exists between the components, stub code specific to the function is invoked to marshal arguments as in `(b)`, and the kernel is invoked. The existence of the kernel-level invocation capability, `KCap`, is verified and the appropriate hardware modifications are made for the current isolation level (e.g. switching protection domains). An upcall to a stub in component 2 then unmarshals the arguments and calls `server_fn`. Returning from the `server_fn` reverses this process. Each invocation increments a counter in the appropriate capability. These counters ease estimation of communication costs across specific isolation boundaries.

The user capability provides discretionary access control, but the kernel capability in conjunction with hardware protection enforces mandatory memory access control when isolation is required. For example, if the user capability is manipulated to reference the `server_fn` directly when a higher level of isolation is present, an invocation of that function would yield a hardware memory access fault.

To alter the isolation level between components, the function pointer in the `UCap`, and the isolation level within the `KCap` have to be updated for each inter-component invokable function. Additionally, the page tables for components must be updated to represent the desired configuration which usually entails two data-cache accesses. Altering the isolation level for a given isolation instance therefore has predictable overhead.

In our prototype implementation on a Pentium 4 2.4 Ghz machine, invocations using `(a)` take 55 cycles averaged over 10,000 runs. This is in contrast to 18 cycles for a virtual function call. An invocation using `(b)` to a function with no arguments takes 1510 cycles (0.63 μsec), 75% of which is incurred by hardware overheads. These numbers are competitive but may be improved with future optimizations.

4 Experimental Evaluation

This section describes a series of simulations involving single-threaded tasks on an Intel Core2 quadcore 2.66 Ghz machine with 4GB of RAM. For all the following cases, isolation benefit for each isolation instance is chosen uniformly at random in the range $[0, 255]$ [1] for the highest isolation level, and linearly decreases to 0 for the lowest level. Unless otherwise noted, the results reported are averaged across 25 randomly generated system configurations, with 3 isolation levels ($\forall i, N_i = 3$), and 3 task constraints (i.e., $1 \leq k \leq 3$). With the exception of the results in Figures 6 and 7, the surplus resource capacity of the knapsack is 50% of the total resource cost of the system with maximum isolation. The total resource cost with maximum isolation is 10000 [2].

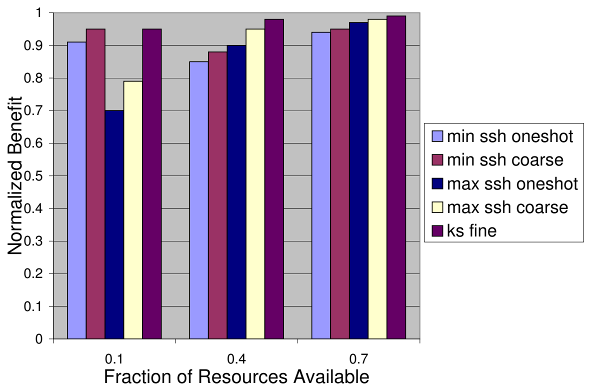
4.1 MMKP Solution Characteristics

In this section we investigate the characteristics of each of the MMKP heuristics. The dynamic programming solution is used where possible as an optimal baseline. We study both the quality of solution in terms of benefit the system accrues and the amount of run-time each heuristic requires. The efficiency of the heuristics is important as they will be run either periodically to optimize isolation, or on demand to lower the costs of isolation when application constraints are not met. In the first experiment, the system configuration is as follows: $|E| = 50$ and the resource costs for each edge are chosen uniformly at random such that $\forall i, k, c_{iN_ik} \in [0, 6]$. Numerically lower isolation levels have non-increasing random costs, and the lowest isolation level has $\forall i, k, c_{i1k} = 0$.

Figure 4(a) shows the normalized benefit of each heuristic with respect to the dynamic programming optimal. The x-axis represents the fraction of the maximal usable resource surplus used as the knapsack capacity. The `ks fine` approach uses the heuristic defined in Section 2.3. Heuristics prefixed with `max` start with maximum component isolation, while those prefixed with `min` start with minimal isolation. Generally, the `ks fine` algorithm achieves

[1]Isolation benefit has no units but is chosen to represent the relative importance of one isolation level to another in a range of [0..255], in the same way that POSIX allows the relative importance of tasks to be represented by real-time priorities.

[2]Resource costs have no units but since we focus on CPU time in this paper, such costs could represent CPU cycles.



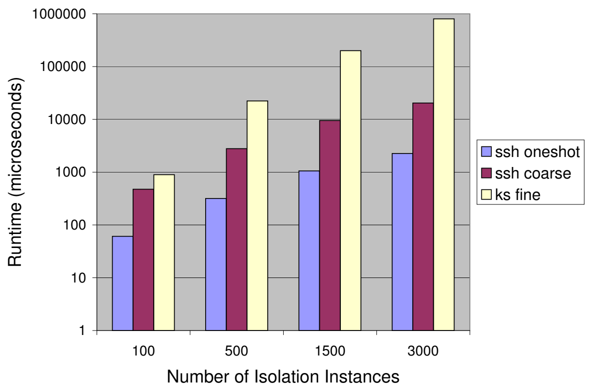


Figure 4. MMKP solution characteristics: (a) MMKP benefit, and (b) heuristic run-times.

high benefit regardless of knapsack capacity. The other algorithms achieve a lower percentage of the optimal when they must alter many isolation levels, but the `coarse` refinement versions always achieve higher benefit relative to the `oneshot` approaches. Altering the number of edges does not affect the results significantly, except for very a small number of edges, so we omit those graphs.

Figure 4(b) plots the execution times of each heuristic while varying the number of edges in the system. The dynamic programming solution does not scale past 50 edges for 3 task constraints, so is not included here. The `oneshot` algorithms' run-times are dominated by sorting, while all `coarse` algorithms demonstrate a higher constant overhead. Contrarily, the `ks fine` refinement heuristic takes significantly longer to complete because of its higher asymptotic complexity.

4.2 System Dynamics

In the following experiments, unless otherwise noted, a system configuration is generated where $|E| = 200$. Resource costs are chosen uniformly at random as follows: $\forall i, k, c_{i3k} \in [0, 100)$, $c_{i2k} \in [0, c_{i3k}]$, and $c_{i1k} = 0$. Note that using resource costs chosen from a bi-modal distribution to model a critical path (i.e., much communication over some isolation boundaries, and little over most others) yield similar results.

Dynamic Communication Patterns: The first dynamic system behavior we investigate is the effect of changing communication patterns between components within the system. Altering the amount of functional invocations across component boundaries affects the resource costs for that isolation instance. Thus, we altered 10% of the isolation instance (i.e., edge) costs after each reconfiguration by assigning a new random cost. All algorithms are able to maintain approximately the same benefit over time. Table 1 shows the percentage of isolation instances that have their isolation weakened, averaged over 100 trials.

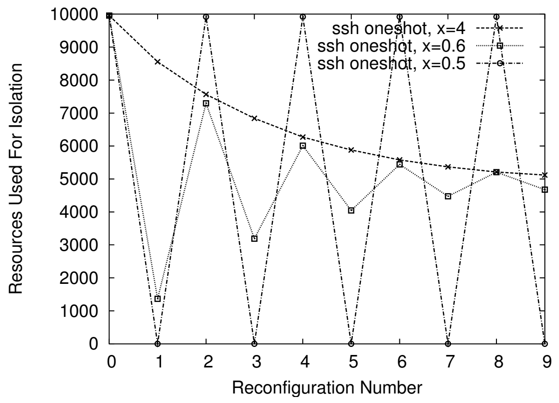
Algorithm	Isolation Instances with Weakened Isolation
`ks oneshot`	3.4%
`ks coarse`	4.2%
`ssh oneshot`	2%
`ssh coarse`	2.5%
`ks fine`	3%

Table 1. Effects of changing communication costs.

Misprediction of Communication Costs: As previously discussed in Section 2.6, misprediction of the cost of communication over isolation boundaries can lead to slow convergence on the target resource availability, or even instability. We use the analytical model in Section 2.6 to predict and, hence, correct isolation costs. This is done conservatively, as Equation 7 assumes that overheads unrelated to isolation hold constant. However, in a real system, factors such as different execution paths within components cause variability in resource usage. This in turn affects the accuracy of Equation 7. Given this, we (1) place more emphasis on predictions made where the difference between the previous and the current resource surpluses is large, to avoid potentially large misprediction estimates due to very small denominators in Equation 7, and (2) correct mispredictions by at most a factor of 0.3, to avoid over-compensating for errors. These two actions have the side-effect of slowing the convergence on the target resource usage, but provide stability when there are changes in resource availability.

Figure 5(a) shows the resources used for isolation by the `ssh oneshot` policy, when misprediction in isolation costs is considered. Other policies behave similarly. In Figure 5(b), the initial misprediction factor, x, is corrected using the techniques discussed previously. The system stabilizes in situations where it does not in Figure 5(a). Moreover, stability is reached faster with misprediction corrections than without.

Dynamic Resource Availability: In Figure 6(a), the light dotted line denotes a simulated resource availability for task $\tau_k \mid k = 1$. The resources available to τ_2 deviate by half as much as those for τ_1 around the base case of 5000. Finally, resource availability for τ_3 remains constant at 5000. This variability is chosen to stress the aggregate resource cost computation. Henceforth, traditional



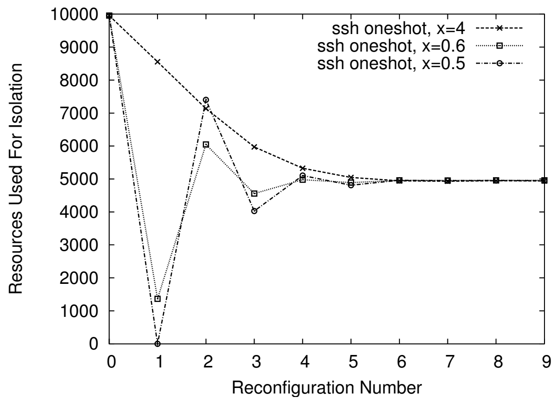
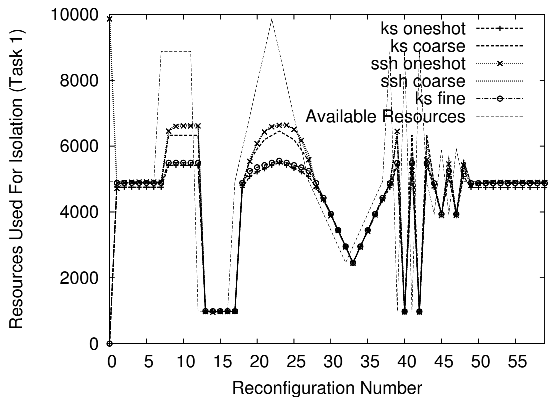


Figure 5. Resources used (a) without correction, and (b) with correction for misprediction costs.



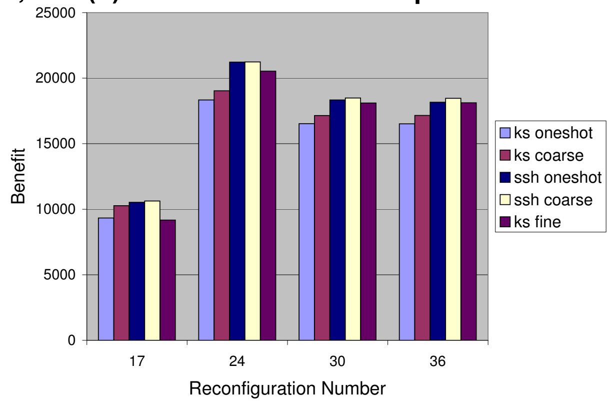


Figure 6. Dynamic resource availability: (a) resources consumed by τ_1, and (b) system benefit.

knapsack solutions that start with minimal isolation will be denoted by `ks`. Consequently, we introduce the `ks oneshot` and `ks coarse` heuristics that behave as in Algorithms 2 and 3, respectively, but compute a system configuration based on an initially minimal isolation state. We can see from the graph that those algorithms based on `ssh` and `ks coarse` are able to consume more resources than the others, because of a more accurate computation of aggregate resource cost. Importantly, all algorithms adapt to resource pressure predictably. Figure 6(b) shows the total benefit that each algorithm achieves. We only plot reconfigurations of interest where there is meager resource availability for τ_1 (in reconfiguration 17), excess resource availability for τ_1 (in 24), a negative change in resource availability (in 30), and a positive change in resource availability (in 36). Generally, those algorithms based on `ssh` yield the highest system benefits, closely followed by `ks fine`.

Combining all Dynamic Effects: Having observed the behaviors of the different algorithms under each individual system dynamic, we now consider the effects of them combined together. Here, we change the cost of 10% of the isolation instances in the system, while the resource availability is changed dynamically in a manner identical to the previous experiment. We assume an initial misprediction factor of $x = 0.6$. Additionally, we employ a conservative policy in which the algorithms only attempt to use 30% of all surplus resources for each reconfiguration.

Figure 7(a) presents the resource usage of task τ_3. Resource availability is again denoted with the light dotted line. Figure 7(b) presents the resource usage for τ_1. Due to space constraints, we omit τ_2. In both cases, the `ssh` algorithms are able to use the most available resource, followed closely by `ks coarse`. The key point of these graphs is that all heuristics stay within available resource bounds, except in a few instances when the resource usage of the current system configuration briefly lags behind the change in available resources. Figure 7(c) plots the system benefit for the different algorithms. As in Figure 6(b), we plot only reconfigurations of interest. In most cases, algorithms based on `ssh` perform best, followed by `ks fine`. Of notable interest, the `ssh oneshot` algorithm generally provides comparable benefit to `ssh coarse`, which has an order of magnitude longer run-time. Figure 7(d) shows the amount of reconfigurations the different algorithms make, that lessen isolation in the system. Although we only show results for several reconfigurations, `ssh oneshot` performs relatively well considering its lower run-time costs.

Next, the feasibility of mutable protection domains is demonstrated by using resource usage traces for a blob-detection application, which could be used for real-time

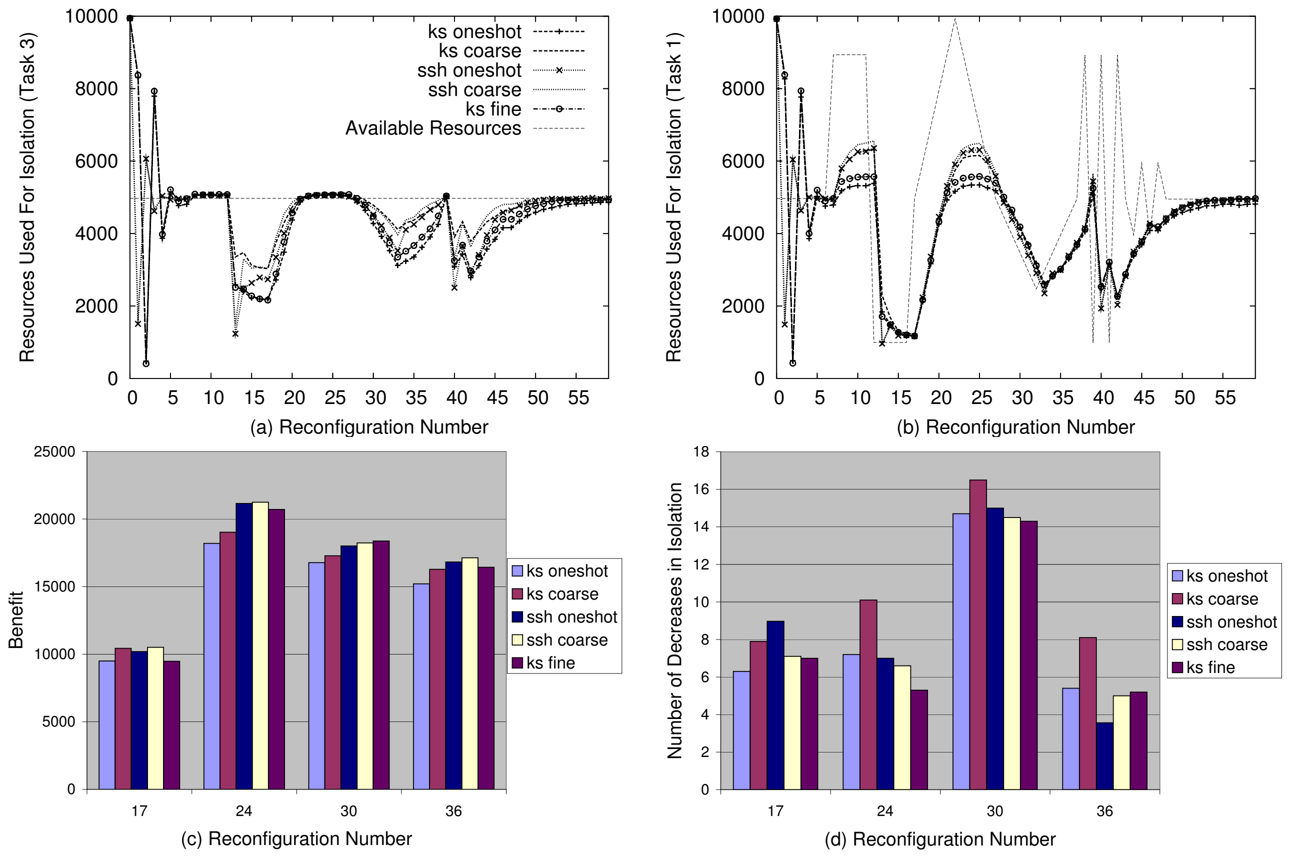


Figure 7. Solution characteristics given all system dynamics.

vision-based tracking. The application, built using the opencv library [13] is run 100 times. For each run, the corresponding execution trace is converted to a resource surplus profile normalized over the range used in all prior experiments: [0,10000]. We omit graphical results due to space constraints. 17.75% of components maintain the same fault isolation for all 100 system reconfigurations, while 50% maintain the same isolation for at least 15 consecutive system reconfigurations. This is an important observation, because not all isolation instances between components need to be changed at every system reconfiguration. On average, 86% of available resources for isolation are used to increase system benefit. Over the 100 application trials, task constraints are met 75% of the time. 97% of the time, resource usage exceeds task constraints by no more than 10% of the maximum available for isolation.

5 Related Work

The multi-dimensional multiple choice knapsack problem (MMKP) has been addressed by others [9, 15, 2]. Similarly, the work on QRAM proposes solutions to the multi-resource discrete QoS problem to maximize some notion of system utility [10]. In other related work, QoS architectures have been developed for the purposes of meeting the needs of real-time and multimedia applications [1, 19, 3]. A key characteristic of our work is the adaptation of system structure, in response to changing resource availability, to maximize isolation benefit while meeting task constraints. A novel aspect covered by our work is the method of addressing mispredictions in the costs of isolation and, hence, communication between components due to e.g. caching.

OS-provided fault isolation has been studied extensively in past research. μ-kernels isolate all but the most fundamental services at user-level, and use efficient IPC to communicate between protection domains [11]. Component-based systems such as Pebble [7] focus on reducing IPC overheads between fine-grained components, while others map components to the same protection domain [12], or support a statically configurable system structure [6, 5]. Other research has produced mechanisms that enhance fault-isolation in monolithic systems [4, 20]. In contrast, this paper proposes an approach that allows the structure of a system to change according to application constraints, with the objective of maximizing fault isolation benefit.

6 Conclusions and Future Work

This paper describes a component-based system supporting *mutable protection domains* (or MPDs). Using MPDs, a system is able to adapt the fault isolation between software components, thereby increasing its dependability at the potential cost of increased inter-component communication overheads. Such overheads impact a number of resources, including CPU cycles, thereby affecting the predictability of a system. We show how such a system can be represented as a multi-dimensional multiple choice knapsack problem (MMKP). Although prior solutions exist for MMKPs, they either require expensive off-line calculations or assume knapsacks are initially empty when deriving a solution. Such empty knapsacks correspond to a system with no (or the lowest level of) component isolation. However, for a practical system to support the notion of mutable protection domains, it would be beneficial to make the fewest possible changes from the current system configuration to ensure resource constraints are being met, while isolation benefit is maximized.

We compare several MMKP approaches, including our own successive state heuristic (`ssh`) algorithms. Due primarily to its lower run-time overheads, the `ssh oneshot` algorithm appears to be the most effective in a dynamic system with changing component invocation patterns, changing computation times within components, and misprediction of isolation costs. The misprediction of isolation costs is, in particular, a novel aspect of this work. In practice, it is difficult to measure precisely the inter-component communication (or isolation) overheads, due to factors such as caching. Using a recurrence relationship that considers misprediction costs, we show how to compensate for errors in estimated overheads, to ensure a system converges to a target resource usage, while maximizing isolation benefit. We are currently building a system using MPDs and future work will focus on the use of various hardware techniques to vary isolation between components.

References

[1] T. F. Abdelzaher and K. G. Shin. End-host architecture for QoS-adaptive communication. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Denver, Colorado, June 1998.

[2] M. M. Akbar, E. G. Manning, G. C. Shoja, and S. Khan. Heuristic solutions for the multiple-choice multi-dimension knapsack problem. In *ICCS '01: Proceedings of the International Conference on Computational Science-Part II*, pages 659–668, London, UK, 2001. Springer-Verlag.

[3] C. Aurrecoechea, A. Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia Systems Journal, Special Issue on QoS Architecture*, 1997.

[4] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Symposium on Operating Systems Principles*, pages 140–153, 1999.

[5] J. Fassino, J. Stefani, J. Lawall, and G. Muller. Think: A software framework for component-based operating system kernels. In *Proceedings of Usenix Annual Technical Conference*, June 2002.

[6] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Symposium on Operating Systems Principles*, pages 38–51, 1997.

[7] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble component-based operating system. In *Proceedings of Usenix Annual Technical Conference*, pages 267–282, June 2002.

[8] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*. ACM, October 1997.

[9] M. S. Khan. *Quality adaptation in a multisession multimedia system: model, algorithms, and architecture*. PhD thesis, University of Victoria, 1998. Advisers: Kin F. Li and Eric G. Manning.

[10] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen. A scalable solution to the multi-resource QoS problem. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 315, Washington, DC, USA, 1999. IEEE Computer Society.

[11] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles*. ACM, December 1995.

[12] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click modular router. In *Symposium on Operating Systems Principles (SOSP)*, pages 217–231, 1999.

[13] OpenCV: http://opencvlibrary.sourceforge.net/.

[14] G. Parmer and R. West. Hijack: Taking control of COTS systems for real-time user-level services. In *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2007)*, April 2007.

[15] R. Parra-Hernandez and N. J. Dimopoulos. A new heuristic for solving the multichoice multidimensional knapsack problem. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 35(5):708–717, 2005.

[16] T. A. R. Wahbe, S. Lucco and S. Graham. Software-based fault isolation. In *Proceedings of the 14th SOSP*, Asheville, NC, USA, December 1993.

[17] Y. Toyoda. A simplified algorithm for obtaining approximate solutions to zero-one programming problems. *Management Science*, 21:1417–1427, 1975.

[18] V. Uhlig, U. Dannowski, E. Skoglund, A. Haeberlen, and G. Heiser. Performance of address-space multiplexing on the Pentium. Technical Report 2002-1, University of Karlsruhe, Germany, 2002.

[19] R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, and D. Bakken. QuO's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, September 1998.

[20] R. West and G. Parmer. Application-specific service technologies for commodity operating systems in real-time environments. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006)*, pages 3–13, 2006.