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

December 6, 2007
Complexity of Embedded Systems

Traditionally simpler software stack
- limited functionality and complexity
- focused application domain

Soon cellphones will have
- 10s of millions of lines of code
- downloadable content (with real-time constraints)

Trend towards increasing complexity of embedded systems
Run-time interactions are difficult to predict and can cause faults

- accessing/ modifying memory regions unintentionally
- corruption data-structures
- deadlocks/ livelocks
- race-conditions
- ...

Faults can cause violations in correctness and predictability
Given increasing complexity, system design must anticipate faults

Memory fault isolation: limit scope of adverse side-effects of errant software

- identify and restart smallest possible section of the system
- recover from faults with minimal impact on system goals
- employ software/hardware techniques

Preserve system reliability and predictability in spite of misbehaving and/or faulty software
Trade-offs in Isolation Granularity

- Increased Isolation
- Reduced Communication Cost

Protection Domains
- Threads
- Stacks

Components

Process Isolation  User-Kernel Isolation  Library Isolation
What is the “best” isolation granularity?

Monolithic OSs

- provide minimal isolation to allow process independence
- large kernel not self-isolated, possibly extendable

→ Coarse-grained isolation, but low service invocation cost
What is the “best” isolation granularity?

\[\text{Net} \quad \text{FS} \quad \text{P1} \quad \text{P2}\]

user-level

kernel-level

IPC

\(\mu\)-kernels

- segregate system services out of the kernel, interact w/ Inter-Process Communication (IPC)
- finer-grained isolation
  - IPC overhead limits isolation granularity

\(\rightarrow\) Finer-grained fault isolation, \textbf{but} increased service invocation cost
What is the “best” isolation granularity?

\[ \text{Net} \quad \text{FS} \quad \text{P1} \quad \text{P2} \]

user-level

kernel-level

IPC

\( \mu \)-kernels
- segregate system services out of the kernel, interact w/ Inter-Process Communication (IPC)
- finer-grained isolation
  - IPC overhead limits isolation granularity

→ Finer-grained fault isolation, but increased service invocation cost

Both characterized by a static system structure
Mutable Protection Domains (MPDs)

- *dynamically* place protection domains between components in response to
  - communication overheads due to isolation
  - application deadlines being satisfied
- application close to missing deadlines
  \[\rightarrow\] lessen isolation between components
- laxity in application deadlines
  \[\rightarrow\] increase isolation between components
Mutable Protection Domains appropriate for soft real-time systems

Protection domains can be made immutable where appropriate
System is a collection of *components*

Arranged into a directed acyclic graph (DAG)
- nodes = components themselves
- edges = communication between them, indicative of control flow

Isolation over an edge can be configured to be one of the three isolation levels
Isolation cost and benefit

Isolation between components causes a performance penalty due to:

1. processing cost of a single invocation between those components
2. the frequency of invocations between those components

→ cost of each isolation level/edge

Different isolation levels yield higher dependability

- stronger isolation → higher dependability

Isolation between specific components more important

- debugging, testing, unreliable components, ...

→ benefit of each isolation levels/edge
Isolation cost and benefit

Isolation between components causes a performance penalty due to:

1. processing cost of a single invocation between those components
2. the frequency of invocations between those components

→ cost of each isolation level/edge

Different isolation levels yield higher dependability

- stronger isolation → higher dependability

Isolation between specific components more important

- debugging, testing, unreliable components, . . .

→ benefit of each isolation levels/edge

This paper studies the policies concerning when and where isolation should be present
Problem Definition

For a solution set $s$

where $s_i \in \{1, \ldots, \#\text{isolation\_levels}\}$
For a solution set $s$

Maximize the dependability of the system . . .

where $s_i \in \{1, \ldots, \#\text{isolation levels}\}$

maximize

$$\sum_{\forall i \in \text{edges}} \text{benefit}_{is_i}$$
For a solution set \( s \) where \( s_i \in \{1, \ldots, \# \text{isolation levels}\} \):

Maximize the dependability of the system \( \ldots \)  

\[
\text{maximize} \quad \sum_{\forall i \in \text{edges}} \text{benefit}_{is_i}
\]

While meeting task deadlines \( \ldots \)  

\[
\text{while} \quad \sum_{\forall i \in \text{edges}} \text{cost}_{is_i} \leq \text{surplus resources}
\]
Problem Definition

For a solution set \( s \)

Maximize the dependability of the system . . .

While meeting task deadlines . . .

For each task in the system

where \( s_i \in \{1, \ldots, \# \text{isolation \_levels}\} \)

maximize

\[
\sum_{\forall i \in \text{edges}} \text{benefit}_{is_i}
\]

while

\[
\sum_{\forall i \in \text{edges}} \text{cost}_{is_i k} \leq \text{surplus \_resources}_k
\]

\( \forall k \in \text{tasks} \)
Multi-Dimensional, Multiple-Choice Knapsack

maximize \[ \sum_{i \in \text{edges}} \text{benefit}_{is_i} \]

subject to \[ \sum_{i \in \text{edges}} \text{cost}_{is_i,k} \leq \text{surplus\_resources}_k, \forall k \in \text{tasks} \]

\[ s_i \in \{1, \ldots, \text{max\_isolation\_level}\}, \forall i \in \text{edges} \]

This problem is a multi-dimensional, multiple-choice knapsack problem (MMKP)

- multi-dimensional - multiple resource constraints
- multiple-choice - configure each edge in one of the isolation levels

NP-Hard problem

- heuristics, pseudo-poly dynamic prog., branch-bound
Effective and inexpensive greedy solutions to one-dimensional knapsack problem exist

- sort isolation levels/edges based on benefit density, ratio of benefit to cost
- increase isolation by including isolation levels/edges from head until resources are expended
- ...but we have multiple dimensions of cost
Compute an *aggregate cost* for each edge

- single value representing a combination of the costs for all tasks for an edge: \( \forall k, \text{cost}_{isik} \rightarrow \text{agg}\_\text{cost}_{isi} \)
- some tasks very resource constrained, some aren’t
- intelligently weight costs for task \( k \) to compute aggregate cost
1. compute aggregate cost for each isolation level/edge
2. include isolation level/edge with best benefit density in solution configuration
3. goto 1 until resources expended

*Fine-grained* refinement of aggregate cost
- recompute once every time an isolation level/edge is added to the current solution configuration
Solutions - *coarse* and *oneshot* Refinement

1. compute aggregate cost for each isolation level/edge
2. sort by benefit density
3. include isolation level/edge from head
4. goto 3, until resources expended
5. recompute aggregate costs based on resource surpluses with solution configuration
6. goto 2 $N$ times and return highest benefit configuration

$N > 1$: *coarse-grained* refinement
- recompute once per total configuration found
- execution time linearly increases with $N$

$N = 1$: *oneshot*
- very quick
- no aggregate cost refinement
Solution Runtimes

Parmer, West, BU CS Mutable Protection Domains
System Dynamics

System is dynamic

- changing communication costs over edges as threads alter execution paths between components
- changing resource availabilities as threads vary intra-component execution time
- per-invocation cost overheads vary
  - different cache working sets, invocation argument size,
  ... 

System must refine the system isolation configuration as these variables change
Solutions over time

System dynamics require re-computation of system configuration

1. disregard current system state, recompute entirely new system configuration
   - traditional knapsack (MMKP) approach: $ks$

2. solve for the next system configuration starting from the current system configuration

*Successive State Heuristic (ssh)*

- modifies *coarse* and *oneshot* to start from the current system configuration
- aim to reduce isolation changes to existing configuration
Experimental Simulations

Simulate a system with

- widely varying resource surplus for 3 tasks
- changing communication costs
- 200 edges, 3 isolation levels
Resource Usage for Task 1

Parmer, West, BU CS
Mutable Protection Domains
System Isolation-Derived Benefit

![Graph showing benefit vs reconfiguration number for different reconfiguration types: ks oneshot, ssh oneshot, ks coarse, ssh coarse, ks fine.](image)
**Composite**: component-based OS designed to support MPD

![Diagram of Composite OS Support for MPD](image)
**Composite**: component-based OS designed to support MPD
Switching between the two isolation levels requires changing UCap, KCap, and protection domains.

Prototype running on x86 Pentium IV @ 2.4 Ghz

- Invocation via kernel - 1510 cycles (0.63 µsecs)
- Direct invocation - 55 cycles (0.023 µsecs)
Conclusions

Solution to MMKP based on lightweight successive refinement given dynamic changes in system behavior

- possibly useful in e.g. QRAM

Mutable Protection Domains

- dynamically reconfigure protection domains to maximize fault isolation while meeting application deadlines
- makes the performance/predictability $\leftrightarrow$ fault isolation tradeoff explicit