

# Quantitative Analysis of Protection Options

Arindam Banerji, Vsevolod V. Panteleenko, Geoff Wyant and David L. Cohn

Technical Report TR-96-20  
Dept. of Computer Science and Engineering  
University of Notre Dame  
Notre Dame, IN 46556

## Abstract

*Various mechanisms are available for protecting software services. Some rely on software protection, some on hardware and some on both. There has been much discussion about protection options, but there is surprising little hard data on relative costs. This paper provides detailed measurements of five different protection schemes and quantifies their impact on performance. These measurements lead to five assertions regarding protection. The cost of hardware-based protection is proportional to the number of transitions from the client to the service. For software-based protection, cost is proportional to the number of instructions executed in the service domain. The effect of protection on the memory subsystem is relatively unimportant. The impact of sharing data between client and service is important for sub-page size data granularities but not for larger data granularities. The evolution of technology will favor hardware-based protection schemes over software-based schemes.*

## 1. Introduction

Operating systems, data bases and other software systems are often called on to isolate untrusted clients from sensitive services. Normally this is done by providing separate protection domains for clients and services, and requiring formal communication between them. This prevents a misbehaving or malicious client from learning sensitive information, damaging data or corrupting the service. It also encourages modular service design for improved flexibility, maintainability, reliability and debuggability.

Unfortunately, sending messages between protection domains can be quite costly. In some cases, the time required to cross the protection boundary is more than that needed to render the service. The cost of protection can be a determining factor in the selection of system software structure. A number of protection approaches have been proposed, and each has its own cost trade-offs. One may be faster for a given system and benchmark mix, but slower on another. More importantly, the continuing evolution of hardware and

software technology will affect different protection options differently. Thus, it is important to understand the nuances of protection overhead in order to make design decisions for the long term.

This paper evaluates five different protection schemes with six benchmark tests. The protection schemes include classic and proposed approaches. Three are hardware-based and depend on the kernel-level protection boundary. The other two use software-based methodologies to assure correct operation.

The benchmarks vary from simple services which just touch the client data to fairly complex computational tasks. The test facilities were unable to properly measure the cost of floating point operations [Welbon 96], so no numerically intensive service was evaluated.

For each scheme, the total number of machine cycles consumed in various operations was measured and analyzed. The results support the following five assertions:

- For hardware-based protection, the cost of protection is proportional to the number of transitions from the client domain to the service domain.
- For software-based protection, the cost of protection is proportional to the number of instructions executed in the unprotected version of the service.
- The effect of protection on the memory subsystem is not very significant.
- It can be valuable to share data between client and service for data sizes up to a page size, but it is relatively less important for larger data sizes.
- The performance threshold between hardware-based and software-based protection depends on the amount of work done in the service, but the evolution of technology will favor hardware-based protection schemes.

The first two assertions are not surprising. The third is different from what earlier work [Chen 93] [Mogul 91] [Bersh 92] implies, and is probably a result of changes in hardware architectures. The fourth is counter intuitive since it seems to say that sharing is better than copying for *small* data objects, but not for larger ones. The final one is most significant and, perhaps, most controversial.

## **2. Protection**

Six different protection schemes are analyzed in this paper. They run the gamut of software-based and hardware-based approaches. The goal was to determine which characteristics of protection had the biggest impact on performance.

Software-based protection is implemented by either applying software fault isolation to the test application and the library, or by writing the entire application and library in Modula3 [Nelson 91]. For hardware-based protection schemes, the entire library is embedded in the kernel, encapsulated as a process or built as a protected shared library (PSL) [Banerji 94].

### **2.1 No protection**

To establish a baseline, all tests were conducted using no protection between the client and service domain. The service was built as a normal procedure, and service invocation was a simple procedure call.

### **2.2 Kernel-based protection**

The classic way to protect service code is to hide it behind the protection boundary that separates the user domain from the kernel domain. User-level code can only enter the kernel domain through carefully controlled entry points, and this prevents problems. For our tests, we made the service implementation a part of the kernel and allowed client access through a system call interface. As with normal kernel-level services, data is copied into the kernel space from user space.

### **2.3 Process-based protection**

One of the goals of the microkernel system structure is to separate services from the kernel. Services are built as user-level server processes, which provides better flexibility, but reduce performance. The performance penalty can be mitigated by migrating the client thread into the service, but it is still costly. Normally, data is copied from the client process to the server process.

## 2.4 Library-based protection

A new approach to protection is to use protected shared libraries. Service are built as shared libraries, but when a client thread invokes a library, its access domain is changed. When in the library, it can access service state information as well as client state information. It is even possible to have state information that is specific to a particular client/service pair.

## 2.5 Language-based protection

Type-safe languages can protect a service from client misbehavior [Bersh 95]. For this example, client and service are implemented as separate modules in Modula-3. Data is freely shared between client and service, and the language guarantees that only legal operations are performed.

## 2.6 Software fault isolation

Rather than constraining programmers to a type-safe language, software fault isolation [Wahbe 93] prevents normal code from violating domain boundaries. Jump and store instructions with static addresses are checked before loading. Dynamic jumps and stores are *sandboxed* within run-time checks which ensure that no illegal accesses are possible. As with language-based protection, data is shared between client code and service code.

## 3. Related Work

Quite a few researchers have analyzed the performance of classic kernel-based protection mechanisms [Anders 91]. [Ouster 90] reports that although cpu performance has improved significantly over the last few years, operating system performance, especially context switch times have clearly not kept pace. His work did not analyze the reasons for this behavior. The impact of context switches on the cache performance is analyzed in [Mogul 91]. This work demonstrated that the cost of cache refills can dominate overall cost of a context switch.

Chen and Bershad [Chen 93] analyzed the effect of the system software decomposition on the memory subsystem performance. The work argues that the separation of operating system functionality into a

microkernel and associated user level servers significantly increases number of idle CPU cycles due to the memory stalls. The authors conclude that the cost of crossing between user-level protection domains renders microkernel architectures inferior to classic monolithic designs. Liedtke [Liedtke 95] has disputed this result by demonstrating microkernel implementations that dramatically lower the cost of domain crossings. However, he agrees that the crossing cost is key.

There has been much work on improving the speed of domain crossings [Bersh 90] [Ford 93] and on reducing the number of such crossings [Bogle 94] [Condict 94]. The problem is projected to get worse as hardware optimizations such as pipelining and caching increase the cost of context switches. Some researchers have proposed hardware support [Carter 94] and new software constructs [Banerji 94] [Yarvin 93] to decrease context switch overhead. [Drush 93] came up with a cross-domain data sharing scheme in order to reduce data transfer costs during domain crossings.

In the recent past, several software alternatives to hardware-based protection have been proposed. Software Fault Isolation, one such software mechanism, is reported to introduce low enough overheads to be useful for safe kernel extensions<sup>1</sup> [Wahbe 93]. Yet another alternative is to use strongly typed languages, like Modula3 [Nelson 91] [Bersh 95] and Java [Gosling 96].

Although the overall cost of well-known protection schemes has been reported, in-depth relative analyses of how and why each scheme adds overhead cycles are not available. Work is beginning to appear on this question [Small 96], but a full analysis is still lacking. This makes it difficult to project the performance trade-offs as technology evolves. This paper attempts to fill this gap.

## **4. Experimental Methodology**

### **4.1 Setup**

The measurements reported here were made on an IBM RS/6000 Model 390 [Weiss 94]. The machines characteristics are summarized in Table 1. Except as noted, tests were conducted with a quiescent system.

1. Typically, kernel extensions are unsafe and have the same ability to crash a machine as device drivers.

**Table 1: Test Machine Characteristics**

---

Processor type	IBM POWER2
Clock Rate	66 MHz
Word width	32
No. of CPUs	1
Instruction Cache	32KB, 2-way, 64-byte lines
Data Cache	64KB, 4-way, 128-byte lines
Instruction TLB	128 byte, 2-way
Data TLB	512 bytes, 2-way

## 4.2 Tools

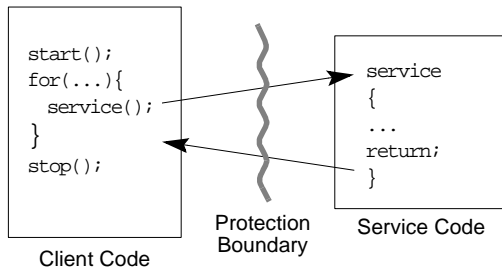
The results reported in this paper are based on hardware and software tools provided by the POWER2 architecture and AIX. The POWER2 architecture has a set of event counter registers [Welbon 95] that can count CPU and memory subsystem events. The measurable events include instructions, machine cycles, cache misses, etc.

The IBM AIX 3.2.5 kernel [IBM 90] is instrumented to trace specific operating system activity, such as system calls and page faults. This trace can be turned on and off. By starting the trace at the beginning of a test and stopping it at the end, a clear picture of OS activity during the test can be obtained. These measurements include the number of page faults, the number and types of system calls, etc.

In addition, for language-based protection we used the DEC-SRC Modula-3 compiler [SRC 96]. To enable SFI, we built our own post-compilation toolset specifically directed towards the Power-2 architecture.

## 4.3 Benchmarks

Five benchmark tests were used to measure the performance of the protection schemes. Two of them involve substantial fixed point arithmetic on large data sets and others entail logical operations on a database. The method of evaluating the benchmarks is described first, and then each of the benchmarks is defined.



**Figure 1:** Client/Service Relationship

**Table 2: Nsieve Problem Sizes**

Number of Primes	Grain Size
1,899	10
2,261	10
4,202	40
7,836	160
14,683	640
27,607	2,560
52,073	10,240
98,609	40,960
187,133	163,840
356,243	655,360
679,460	2,621,440

### Evaluating benchmarks

In each case, the benchmark code is built as a service which is invoked by a client. The goal is to evaluate the cost of protecting the service from the client. Figure 1 shows how each invocation has to cross from client code to service code, and it indicates where we start and stop our data gathering.

The benchmarks each have parameters that adjust the size of the test. We categorize them as follows:

- Invocations ( $I$ ) - The number of times the service is invoked.
- Granularity ( $G$ ) - A measure of the amount of work done in the service for each invocation.
- Problem Size ( $N$ ) - The product  $G$  time  $I$ .

As we shall see, the relationship between these size measures and the number of machine cycles required to complete the test can be quite complex.

### MD5

MD5 is a secure one-way hash function that was developed to reliably identify long byte strings [Rivest 92]. The implementation used here is based on code made available by RSA [RSA 93]. The input byte string is partitioned into fixed-length substrings, and the algorithm successively operates on the substrings. We use substring length as the granularity  $G$  of our test and the input string length  $N$  as the problem

size. Thus, the number of invocations will be  $N/G$ . The first call also invokes an initialization routine, and the last call uses a final routine.

### *Nsieve*

Nsieve is a well-known benchmark that computes prime numbers. It was originally developed to assess the relative performance of computer systems. Our code is based on a public domain version [Aburto 94]. Problem size for Nsieve is the total number of primes to calculate, and granularity is the number of numbers searched. Table 2 gives the eleven standard value pairs. The iterative portion of the standard Nsieve code was built as the service, so the number of invocations depends on the density of the primes.

### *tdbm\_i, tdbm\_f, tdbm\_d*

Three benchmarks involve our tdbm database, a small in-memory database based on the Berkeley UNIX ndbm library. It is a slight modification of the sdbm library released by Ozan Yigit [Yarvin 93], and is based on the 1978 dynamic hashing algorithm by Paul Larson [Enbody 88]. The changes avoid unnecessary copying and remove file dependence. The tests involve insertion of  $N$  words from an extended version of `/usr/dict/words`, random fetch of  $N/2$  words, and deletion of  $N/2$  words. For *tdbm\_i*, problem size is  $N$ ; for *tdbm\_f* and *tdbm\_d*, it is  $N/2$ . In every case, granularity is 1.

### *Nullc*

Nullc is a custom benchmark that is essentially a null call. The service is passed a block of data, it touches every data page and returns the data to the client. This test measures the base cost of transferring variable-sized parameters between protection domains. The granularity  $G$  of the data block varies from 4 bytes to 256Kbytes. For each size, the total number of iterations  $I$  is 512.

## **5. The Cost of Protection**

The performance cost of a protection option is the number of extra machine cycles required to complete a task. This was computed by measuring the number of cycles for unprotected operation and subtracting it

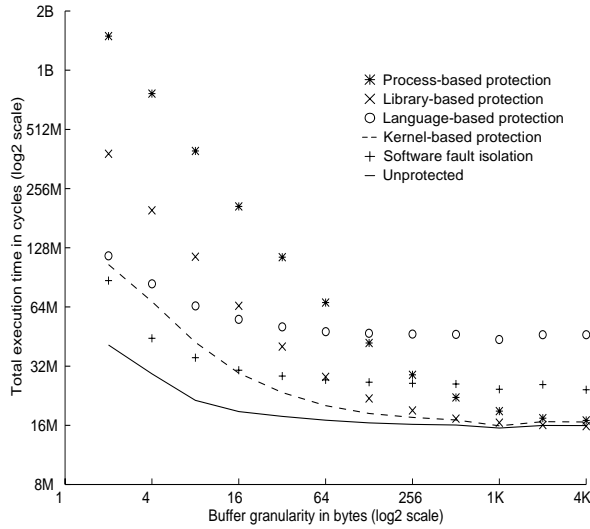


Figure 2: Execution times for md5

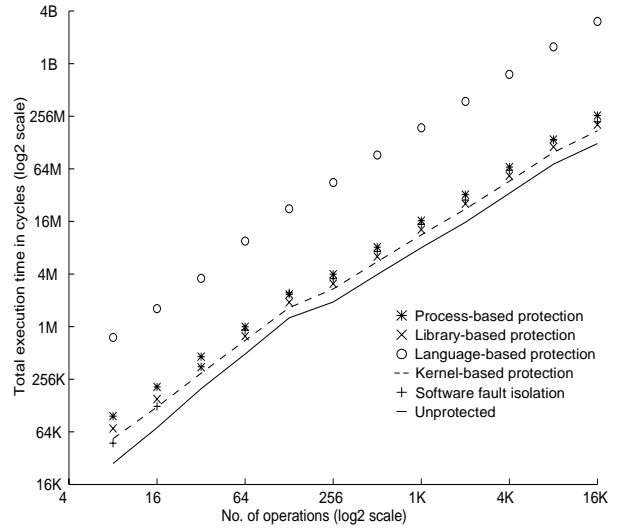


Figure 3: Execution times for tdbm\_d

from the number of cycles using a given protection option. This section presents the raw execution times for various protection schemes and shows how these data validate the first two assertions.

### 5.1 Protection Overhead

For each benchmark, we varied the granularity of the protection domain, and recorded the number of machine cycles needed to perform the service. For the md5 tests, the total message size was kept constant at 512 KBytes, and the number of bytes passed to the service was varied. Figure 2 shows the resulting performance as a function of problem granularity. As the granularity increases, there are fewer service invocations, and the execution time of all schemes decreases.

For the tdbm benchmarks (Figure 3), we vary the number of elements in the database. Each invocation deals with one entry, but for larger databases, the service does more work. These performance curves differ significantly from the md5 curves in Figure 2.

### 5.2 Cost of Hardware-based Protection

The total number of machine cycles needed to execute  $I$  invocations of a service can be decomposed into:

$$C = IT + \sum_{i=1}^I S(i) \quad (1)$$

where it takes  $T$  cycles to transition from the client code to the service code and  $S(i)$  cycles to make the  $i^{\text{th}}$  invocation and provide the service. We have indicated that  $S$  is a function of  $i$  since, for a given problem, the number of cycles in the service code may vary between successive invocations. For simplicity, we will denote the average number of cycles in the service, including the cost of initiating and terminating the invocation, as:

$$\hat{S} = \frac{1}{I} \sum_{i=1}^I S(i) \quad (2)$$

We will use subscripts to distinguish between the number of machine cycles for three cases: unprotected (U), hardware-based protection (H) and software-based protection (S). Thus, we can write:

$$C_H = IT_H + I\hat{S}_H \quad (3)$$

For hardware-based protection, all of the cost should be incurred in transitioning from the client code to the server code. That is, we expect

$$\hat{S}_H = \hat{S}_U \quad (4)$$

Therefore, the total extra cost of hardware-based protection should be a linear function of the number of transitions:

$$C_H - C_U = I(T_H - T_U) \quad (5)$$

Figure 4 illustrates the protection overhead in cycles for the md5 benchmark plotted as a function of the number of service invocations. The plot is on a log-log scale. Except for some small-test effects, protection overhead for the three protection schemes illustrated increases at a  $45^\circ$  angle, indicating a linear relationship. Other benchmark tests had similar characteristics of the overhead. In each case, the offset indicates

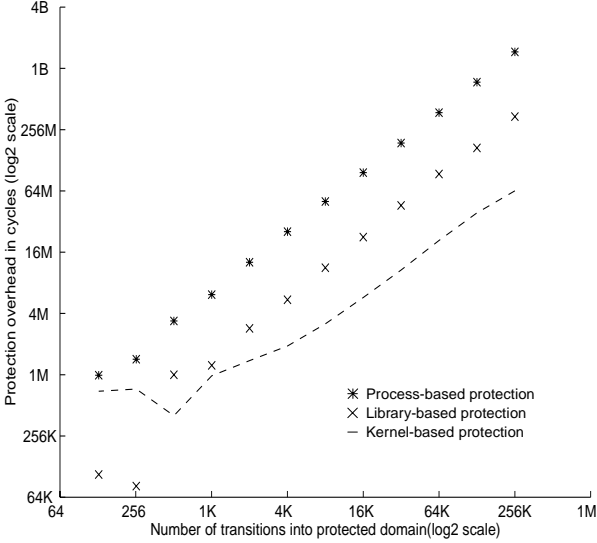


Figure 4: Protection Overhead for md5 - H/W

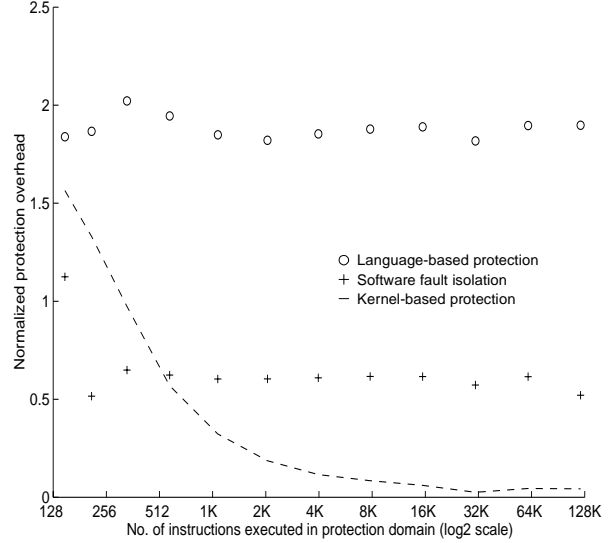


Figure 5: Normalized Protection Overhead - S/W

the constant of proportionality  $\alpha$  between the hardware-protected and unprotected transition cost. Thus, the total hardware-protected cost is:

$$C_H = \alpha IT_U + \hat{I}S_U \quad (6)$$

### 5.3 Cost of Software-based Protection

For software-based schemes, protection overhead is a bit more complex. Software fault isolation adds instructions, and hence cycles, for each store or branch whose address is determined dynamically. The number of such instructions is loosely related to the total number of instructions executed when no protection is used, but there is no deterministic relationship. A similar relation holds for language-based protection. The language constraints have a performance cost that depends loosely on the number of instructions.

We have asserted that for both cases, the protection overhead is essentially linear in the number of cycles. This implies that for software fault isolation, the number of stores and branches with dynamic addresses is essentially proportional to the number of instructions, and that for language-based protection, the overhead is essentially proportional to the number of instructions. For both cases, it implies that the number of instructions is essentially proportional to the number of cycles.

If this assertion is true, the cost of software-based protection can be written as

$$C_S = \beta IT_U + \beta I \hat{S}_U \quad (7)$$

where  $\beta$  is based on the proportionality relationships. To verify Eq (7), note that it implies that the protection penalty for software-based protection can be written:

$$C_S - C_U = (\beta - 1)I(T_U + \hat{S}_U) \quad (8)$$

Dividing by  $C_U$  will normalize this:

$$\frac{C_S - C_U}{C_U} = \beta - 1 \quad (9)$$

In other words, the normalized software-based protection overhead would be a constant.

To verify our second assertion, we have plotted the normalized software-based protection overhead for the md5 benchmark in Figure 5. Both language-based protection and software fault isolation have essentially constant normalized protection overhead. The constant value indicates the average increase in execution time due to the run-time checking. By contrast, the normalized protection overhead for kernel-based protection drops off as problem size increases. Tests for other benchmarks also showed near constant normalized overhead.

## 6. Impact on Memory Subsystem

This section investigates the impact of protection schemes on the memory subsystem. One key measurement of memory subsystem performance is the average number of memory cycle overhead per instruction (MCPI). MCPI effectively measures the average time an instruction spends waiting for memory accesses. Memory accesses often cause the bulk of instruction latency in modern systems. Thus MCPI indicates a given scheme's effect on the cost of execution with respect to memory accesses. Table 3 lists MCPI for the five protection schemes and several example benchmarks. Table 4 shows how MCPI changes for different problem size of the same benchmark, in this case Nsieve.

**Table 3: MCPI for various Benchmarks**

Benchmark		Protection Method					
<u>Test</u>	<u>Size</u>	<u>None</u>	<u>Library</u>	<u>Kernel</u>	<u>Process</u>	<u>SFI</u>	<u>Language</u>
nsieve	14,683	0.128	0.128	0.738	0.138	0.091	0.008
tdbm-i	512	0.021	0.030	0.020	0.031	0.010	0.018
tdbm-f	512	0.030	0.033	0.073	0.033	0.017	0.032
tdbm_d	512	0.010	0.010	0.010	0.010	0.010	0.020
md5	64	0.039	0.039	0.001	0.042	0.020	0.007

Table 3 shows that protection schemes don't generally increase MCPI. We were surprised that both software-based protection schemes (SFI and language-based protection) actually reduced MCPI. Analysis of the code (that is of services protected by SFI and language-based protection) showed that these schemes add many *cheap* instructions; that is, these instructions are fast to decode and make few memory references. Many of these instructions check array bounds and target addresses by comparing arguments already present in on-chip registers.

When operating on small amounts of data, the software-based schemes showed some increased paging, due primarily to increased code-size caused by added software checks. As the amount of data increases, the software and hardware schemes showed similar I/O characteristics. This happens due the fact that paging caused by large data sizes begin to dominate the I/O costs for both software and hardware-based protection schemes.

It is interesting to compare these measurements with the results obtained by Chen and Bershad. In their Mach-based measurements, they saw fairly high MCPI values (as high as 0.7) and concluded that process-based protection had a fairly high cost. In our measurements, we have uniformly low values for MCPI regardless of the protection scheme. Similar low MCPI values for newer processors have also been reported by Rosenblum [Rosenb 95]. We attribute this to the evolution of processor architectures and system implementations between 1993 and 1995. In particular, the system upon which we took our measurements offers non-blocking caches, out-of-order execution and high processor-memory bandwidth

**Table 4: MCPI for Various Sizes of Nsieve**

<u>Problem Size</u>	<u>None</u>	<u>Library</u>	<u>Kernel</u>	<u>Process</u>	<u>SFI</u>	<u>Language</u>
187,133	0.20	0.21	0.18	0.22	0.13	0.01
356,243	0.21	0.22	0.21	0.22	0.14	0.01
679,460	0.22	0.23	0.23	0.23	0.15	0.01

[McVoy 96]. These features have greatly lowered the cost of protection as compared to systems available in 1993<sup>2</sup>.

It is important to note that this result does not suggest that in current and future systems, MCPI values will be low for all programs. It only suggests that the *added* MCPI due to protection will be quite low. Thus, for a software service that already has a high MCPI value of 0.75 (in an unprotected state), adding protection will only increase the MCPI value minimally. The overall MCPI value nonetheless will remain high.

## 7. Effect of Data Access

In addition to the invocation itself, the service typically accesses data and, perhaps, transfers data back to the client. The performance cost of the data access method can be unexpectedly large. Our tests have demonstrated that the common wisdom that says that small data items can be copied but large data items should be shared is misleading.

### 7.1 Copying vs. Sharing

Data access can either be by copying, where data is actually placed in the service domain, or sharing, where the data remains in the client domain. The software-based protection schemes naturally allow sharing, and sharing is possible with all of the hardware-based methods. However, most implementations of sharing

2. The memory subsystem performance of the 1993 POWER-2 based systems is comparable to that of 1996 systems. The contemporary SuperSparc had a 50MB/s throughput and a 800ns latency. The R4000 of the same time had a 60MB/s throughput with 1100ns latency. In comparison, the POWER2 based system used for our measurements had 800MB/s throughput with 220ns latency.

kernel-based and process-based protection copy the data into the kernel rather than accessing it in the client domain. We assume here that when copying is performed, standard optimization techniques provided by the operating system like copy-on-write are used whenever possible.

With protected shared libraries, it is easy to specify whether the service is allowed to access data in the client domain. Thus, we use PSLs as our example protection mechanism for assessing the effect of data access. We ran each test with both copied and shared data, measured the resulting overhead and decomposed this overhead to better understand the cost of data access.

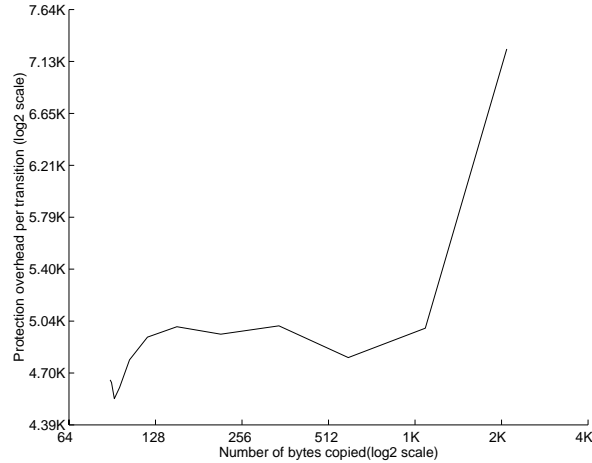
## 7.2 Cost of Copying

If the number of cycles due to copying is a multiple of the number of bytes copied plus an initialization cost, then the difference between number of cycles with copying,  $C_{\text{copy}}$ , and that for sharing,  $C_{\text{share}}$ , could be written:

$$C_{\text{copy}} - C_{\text{share}} = I (GT_{\text{byte}} + T_{\text{init}}) \quad (10)$$

where  $I$  is the number of invocations,  $G$  the number of bytes transferred,  $T_{\text{byte}}$  the transfer cost per byte and  $T_{\text{init}}$  the cost to initialize the transfer.

If Eq (10) were valid,  $(C_{\text{copy}} - C_{\text{share}})/I$  would be linear in  $G$ . A plot of this quantity, for the md5 test and various values of  $G$  is shown in Figure 6. Several factors appear to cause the departure from linearity. For block sizes greater than the cache line size of 128 bytes, the function is essentially flat. Experimental data shows that the increased rate of cache misses (due to larger than 128-byte data) introduces added cache-miss costs over the cost of copying. For cache miss effects to be visible, the number of page faults for the range of data sizes of interest should be constant which is true for this part of the graph. The figure also shows how page faults and TLB misses affect the graph when the data size approaches the 4KByte page size. Analysis indicates that page faults dominate the cost, increasing from 3.5 extra faults for 1112 bytes of data to 4.9 for 2136 bytes (data crosses page boundaries).



**Figure 6:** Normalized Cost of Moving Data - md5

**Table 5: Copying Cost - Cycles per Byte**

Benchmark				Test Time		Copying Overhead			
<u>Test</u>	<u>G</u>	<u>I</u>	<u>Bytes</u>	<u>Copying</u>	<u>Sharing</u>	<u>Total</u>	<u>Page Faults</u>	<u>Memory Stalls</u>	<u>Pipeline Stalls</u>
Nsieve	7,836	512	40,036	82.91	81.71	1.20	0.12	0.07	0.19
tdbm-i	1	512	5,424	3.63	3.02	0.61	0	0.004	0.10
tdbm-f	1	512	5,422	2.72	2.13	0.66	0	0.01	0.10
tdbm-d	1	512	5,294	3.02	2.36	0.66	0	0.005	0.15
md5	512	512	1,112 <sup>1</sup>	64.36	56.32	8.04	1.48	0.63	2.96
nullc	1,024	512	1,024	10.80	1.43	9.37	0.94	0.03	3.00

1. Number of bytes sent to service. Only 88 bytes returned to client.

### 7.3 Cost vs. Data Size

Table 5 shows the cost per byte of copying for each of the six benchmarks. In each case, *I* was 512, and the amount of data transferred depended on the test. For the first four tests, the data size exceeded that of a page, and the difference in cost was one cycle or less per byte transferred. However, for the last two tests, which sent minimal amounts of data, the added cost was an order of magnitude more. Kernel-based protection scheme data showed similar behavior. This seems to imply that it is okay to copy large data blocks but that small data blocks should be shared! A highly counter-intuitive result.

For large data blocks, the underlying system can apply clever techniques to minimize the cost of logical copies. In fact, most of these schemes are effective for data sizes greater than a page size in granularity. Fast block copies, page remapping and copy-on-write significantly speed up the process of moving data across protection domains. However, it must be noted that for blocks less than the page size, these optimizations are generally not applicable. Micro-kernel implementations such as Mach, also apply similar techniques to avoid data copying costs for messages of page-size granularity and greater.

Approximately one fifth of the copying overhead is due to pipeline stalls. For some tests, copying caused extra page faults. There were only a few of these, but they added substantially to the copying cost. By contrast, memory stalls had little effect.

## 8. Implications of Results

Our results imply that the relative efficiency of the protection options depends on  $\hat{S}$ , the average number of cycles needed to provide a service. Section 5 describes the correlation between software protection overhead and the work done in the service. Here we analyze how the evolution of hardware, operating system and compiler technologies will affect protection overhead. We argue for a break-even service size above which hardware-based schemes are preferable.

### 8.1 Cost of Protection vs. Service Size

Using Eq (6) and Eq (7), we can formulate a rule for choosing between hardware- and software-based protection. The hardware-based protection overhead, normalized by the number of service invocations, is  $(\alpha-1)T_U$ . That for software-based protection is  $(\beta-1)(T_U + \hat{S}_U)$ . Thus, a reasonable heuristic for picking hardware-based protection is:

$$\text{if } (\alpha - 1)T_U < (\beta - 1)(T_U + \hat{S}_U) \Rightarrow \text{H/W} \quad (11)$$

Typically, the work done in the service  $S_U$  is much greater than the cost of an unprotected service invocation  $T_U$ , so Eq (11) simplifies to:

$$\text{if } (\alpha - 1)T_U < (\beta - 1)\hat{S}_U \Rightarrow \text{H/W} \quad (12)$$

In other words, we have to compare the transition overhead of hardware-based schemes to the service overhead of software-based approaches.

## 8.2 Hardware-Based Transition Overhead

To assess the impact of operating system and hardware evolution, we will examine the hardware-based transition overhead  $(\alpha-1)T_U$  for library-based protection. As we shall see, most of the transition overhead is due to *aliasing*, the resolution of multiple virtual addresses for the same physical address. The POWER2 architecture provides hardware support for resolving aliases, but AIX 3.2.5, an older version of AIX, does not utilize this support.

### *Aliasing cost*

Aliasing arises with hardware-based protection, because client and service domains are different virtual address spaces. Consequently, virtual memory data structures must be updated when control is transferred between them. Also, references to shared data can cause TLB misses. As we shall see, the resulting *alias faults* dramatically impact transition overhead.

To assess the cost of aliasing, we counted the number of instructions in the transition code for the simplest service, *nullc* with a 4 byte transfer. This code calls AIX routines, written in C, which adjust the virtual memory data structures. Thus, the difference between the transition code instruction count (210) and the total instructions used to make the transition (1250) provides an indication of the cost of aliasing (1040 instructions).

Table 6 lists  $(\alpha-1)T_U$  for six benchmark tests with library-based protection. The value for *nullc* and *md5* are essentially the same, but for the others, it is much higher. To understand why, we have plotted the components of  $(\alpha-1)T_U$  in Figure 7.

**Table 6: Transition overhead**

Benchmark	Size	$(\alpha-1)T_U$
md5	512KB	1351
nsieve	14,683	6319
tdbm_i	512	6446
tdbm_f	512	7208
tdbm_d	512	4833
nullc	1024	1485

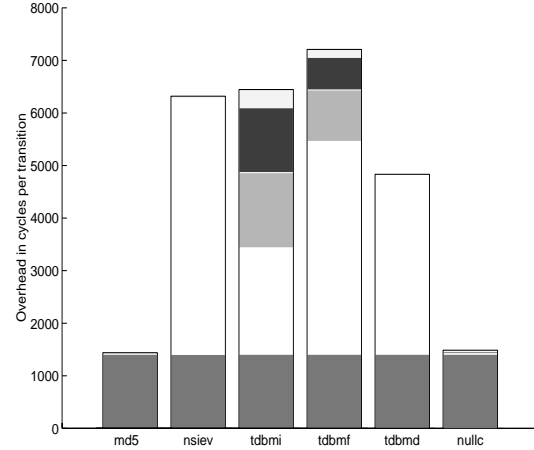
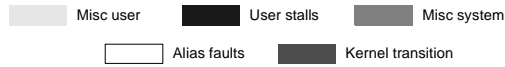
**Figure 7: Library-based Protection Costs**

Table 6 shows that the in-kernel transition cost, which includes the aliasing adjustments, is essentially the same for all benchmarks. For nullc and md5, only one domain actually touches the shared data, so there are no alias faults. However, for the other tests, both client and service access the data, which results in considerable time spent handling aliasing faults. If we set this time aside, we see that for five of the six tests,  $(\alpha-1)T_U$  lies between 1300 and 2500 cycles.

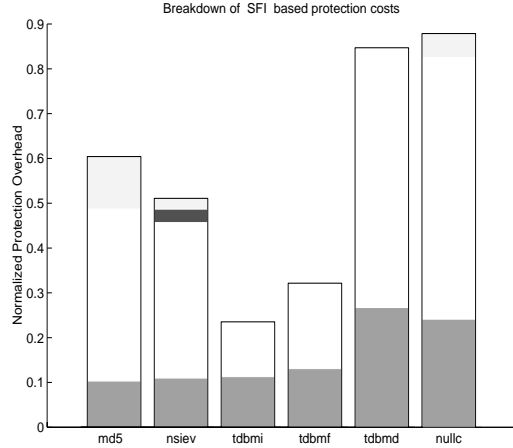
### Trap costs

Since the aliasing problem can be solved in hardware, it is important to look at the other costs. These are approximately 210 instructions per transition, or about 275 cycles. This approximates highly optimized cross-domain transfer times [Hamil 93].

Without aliasing, the hardware cost of the kernel trap and return significant, 57 cycles for POWER2. Library-based protection traps twice for every service invocation accounting for 114 of the remaining 275 overhead cycles. Some modern processors, such as the UltraSparc, have reduced trap overhead to as little as 11 cycles (thus implying a 22 cycle overhead for a PSL-like protection scheme). We expect trap cost to continue to decrease as processors evolve.

**Table 7: SFI-based protection costs**

Benchmark	Size	$(\beta-1)$
md5	512KB	0.61
nsieve	14,683	0.51
tdbm_i	512	0.24
tdbm_f	512	0.32
tdbm_d	512	0.84
nullc	1024	0.88

**Figure 8: SFI-based Protection Costs**

### Data access cost

As we saw in Section 7, data access can be a significant source of cost for hardware-based protection schemes. Increasingly, operating systems are avoiding copying by page remaps and copy-on-write [Drush 93]. Therefore, we have not explicitly broken out copying costs here.

With the impending changes in hardware and operating systems technology, we expect the overhead per transition  $(\alpha-1)T_U$  to drop to about 175 cycles.

### 8.3 Software-Based Protection Schemes

The software-based protection overhead,  $(\beta-1)$ , is sensitive to changes in compiler technology. This section examines possible improvements in  $(\beta-1)$  based on discussions elsewhere in the literature. It uses software fault isolation (SFI) as its example software-based protection scheme.

Table 7 lists the values of  $(\beta-1)$  for several benchmarks using SFI, and Figure 8 plots their decomposition. The largest cost components are the instructions used for fault isolation and pipeline stalls. Improvements in compilers are steadily reducing both of these components. Current hardware trends, namely increase in superscalar width of the processors, can also improve relative performance of software-based protection since overhead instructions can be executed in stolen otherwise pipeline cycles. Finally, special

hardware support for software-based protection like support for bound checking also might be available in the future.

It is difficult to predict the impact of these improvements. Verifiable reports of average  $(\beta-1)$  values lie between 0.18 and 0.22 [Wahbe 93]. Figures reported for a commercial compiler that supports SFI lie in the range of 0.17 to 0.50 [Small 96]. These numbers are comparable to the minimum values of  $(\beta-1)$  we obtained for our benchmarks. Thus, we expect that, with continued improvements in compiler technology, the performance impact of using SFI will be 17%-25%.

Most of the above arguments are also valid for the strongly-typed languages like Modula-3 since they also have the bound checking overhead. At the same time, there is additional high-level information available to the compiler that might allow to avoid some of the overhead instructions that are required in SFI and potentially achieve better performance. Unfortunately in our case, the DEC SRC compiler showed much worse performance than SFI. There are indications that better Modula-3 compiler implementations may remove these problems [Sirer 96].

The other strongly-typed language that we experimented with was Java. An interpreted version of Java with just-in-time compilation optimization was used. The timing tests indicated results an order of magnitude slower than with any other protection scheme which makes it difficult to compare to any of the protection schemes discussed in the paper. In other words, the impact of the (as yet) poor quality of the Java tools is too high to perform any kind of realistic analysis. As an example, md5 Java test with granularity 1024 and the same 512KB problem size took 201 million cycles comparing to 16 million cycles of the unprotected case (in experiments with Java, we used machine with AIX 4.2 operating system since Java is unavailable on AIX 3.2.5). We did not have native code Java compiler but expect that the results will be similar to those for Modular-3.

## 8.4 Break Even Point

The lower-bounds on hardware- and software-based protection overhead allow us to evaluate the break-even point. The break-even point is the average service size  $S_U$  above which hardware-based schemes are preferable to software-based schemes. Using 175 cycles for  $(\alpha-1)T_U$  and 0.17 for  $(\beta-1)$ , we get a break-even point of 1029 cycles. Assuming a CPI near 1, this translates to a break-even service size of about 1000 instructions.

Thus, current trends in technology imply that for service sizes of 1000 instructions or more hardware-based protection schemes are preferable to software-based schemes. Unless the number of added instructions for software-based schemes can be reduced drastically, the cost of software based protection will probably not show a remarkable decline. On the other hand, the cost of hardware based schemes is decreasing steadily. Thus, for most service domain sizes (except really small services) hardware-based schemes will present a better protection option in the future.

## 9. Conclusion

Providing protection is one of the key operating system services. Thus, understanding the performance trade-offs of different protection mechanisms is essential for proper operating system design. There has been little systematic work in analyzing the relative cost of protection mechanisms. In this paper, we provided detailed measurements for a variety of protection mechanisms. Based on these measurements, we provide a set of formulas which can be used to determine when hardware-based mechanisms are more cost-effective than software-based ones. We believe this information is useful when designing protection facilities.

## 10. References

- [Aburto 94] A. Aburto, <ftp://ftp.nosc.mil/pub/aburto/bench.tar.gz>
- [Anders 91] Anderson, T., Levy H., Bershad B., Lazowska E., The interaction of architecture and operating system design, Architectural Support for Programming Languages and Operating Systems, Santa Clara, CA,
- [Banerji 94] A. Banerji, D. Cohn, Protected Shared Libraries, *Tech Rpt 94-37*, Notre Dame, 1994.
- [Bersh 90] Bershad B., Lightweight Remote Procedure Call, *ACM Transactions on Computer Systems*, No. 1, February, 1990

- [Bersh 92] B. Bershad, The Increasing Irrelevance of IPC Performance for Micro-kernel-Based Operating Systems, *Proc Microkernels and Other Kernel Architectures*, USENIX, pp. 205-211, 1992.
- [Bersh 95] B. Bershad, et.al, Extensibility, Safety and Performance in the SPIN Operating System, *Proc. 15th Symp on Operating Systems Principles*, ACM, pp. 267-284, 1995.
- [Bogle 94] P. Bogle, B. Liskov, Reducing Cross Domain Call Overhead Using Batched Futures, *Proc OOPSLA 94*, ACM, 1994.
- [Carter 94] N. Carter, et. al., Hardware Support for Fast Capability-based Addressing, *Proc ASPLOS VI*, ACM, 1994.
- [Condict 94] M. Condic, et. al., Microkernel Modularity with Integrated Kernel Performance, Unpublished Technical Report, OSF, 1994
- [Chen 93] J. Chen, B. Bershad, "The Impact of Operating System Structure on Memory System Performance", *Proc 16th Symp on Operating System Princ*, ACM, pp. 120-130, 1993
- [Drush 93] P. Drushel, L. Peterson, Fbufs: a High Bandwidth Cross-Domain Transfer Facility, *Proc 14th Symp Operating System Principles*, ACM, pp. 189-202, 1993.
- [Enbody 88] R. Enbody, H. Du, Dynamic Hashing Schemes, *ACM Computing Surveys*, Vol.20, No.2, pp. 85-113, 1988
- [Ford 93] B. Ford, J. Lepreau, Evolving Mach 3.0 to a migrating Thread Model, Technical Report, C. S. Dept., Univ Utah
- [Gosling 96] J. Gosling, et. al., The Java Language Specification, *Addison-Wesley*, 1996.
- [Hamil 93] G. Hamilton, P. Kougiouris, The Spring Nucleus: a micro-kernel for objects, *Proc. Summer USENIX conference*, USENIX, pp. 147-159, 1993.
- [IBM 90] *Kernel Extensions and Device Support Programming Concepts - AIX Version 3 for RISC System/6000*, IBM, Austin, TX
- [Liedtke 95] J. Liedtke, On Micro-Kernel Construction, *Proc 15th Symp Operating System Principles*, ACM, pp. 237-250, 1995
- [Mogul 91] J.Mogul, A. Borg, The Effect of Context Switches on Cache Performance, *4th Int'l Conf Architectural Support for Programming Languages and Operating Systems*, ACM, pp. 75-85, 1991
- [McVoy 96] L. Mcvoy, Memory Bandwidth, Usenet news article 48892 of comp.arch, March, 96.
- [Nelson 91] G. Nelson, ed, *System Programming with Modula 3*, Prentice Hall, 1991.
- [Ouster 90] J. Ousterhout, Why Aren't Operating Systems Getting Faster As Fast as Hardware? *Proc USENIX Summer Conference*, USENIX, 1990.
- [Rivest 92] R. Rivest, The MD5 Message-Digest Algorithm, *Network Working Group RCF 1321*, 1992.
- [Rosenb 95] M. Rosenblum, E. Bugnion, S. Herrod, E. Witchel, A. Gupta, A., The Impact of Architectural Trends on Operating System Performance", *Proc 15th Symp Operating System Principles*, ACM, pp. 285-298, 1995
- [RSA 93] <http://www.rsa.com/pub/md5.txt>
- [Sirer 96] Sirer G., Savage S., Pardyak P., Defouw, Alapat M, Bershad B, Writing Operating Systems in Modula-3, <http://www.cs.washington.edu/research/projects/spin/www/m3os.ps>
- [Small 96] C. Small, M. Seltzer, A Comparison of OS Extension Technologies, *Proc USENIX Winter Conference*, USENIX, 1996.
- [SRC 96] <http://www.research.digital.com/SRC/modula-3/html/home.html>
- [Wahbe 93] R. Wahbe, S. Lucco, T. Anderson, S. Graham, Efficient Software-Based Fault Isolation, *Proc 14th Symp Operating Systems Principles*, ACM, pp. 203-216, 1993.
- [Weiss 94] S. Weiss, J. Smith, *POWER and PowerPC*, Morgan Kaufmann, 1994.
- [Welbon 95] E.H. Welbon, C.C. Chan-Nui, D.J. Shippy, and D.A. Hicks, POWER2 Performance Monitor, <http://www.austin.ibm.com/tech/monitor.html>
- [Welbon 96] E.H. Welbon, Personal Communications
- [Yarvin 93] C. Yarvin, et. al., Anonymous RPC: Low Latency Protection in a 64-bit Address Space, *Proc. USENIX Summer Conference*, USENIX, 1993.
- [Yigit 92] O. Yigit, <ftp://ftp.x.org/contrib/util/sdbm>

## 11. Appendix

This appendix presents a table of the benchmark characteristics. The values are for the unprotected version of each test. The tests were run for 10 different problem sizes, and the mean, minimum and maximum values were found. In most cases, only the mean values have been shown here; in others all three are presented.

Data Types		MD5	Nsieve	tdbm-i	tdbm-f	tdbm-d
Cycles (M)	min	16.3	1482	0.126	0.17	S - 0.3
	avg	21.1	1803	65.36	12.65	23.0
	max	43.0	2102	399.3	77.52	130.7
Instructions (M)	min	16.2	1254	0.1	0.013	0.026
	avg	20.1	1563	53.9	11.8	22.5
	max	39.2	1686	327	71.9	126.9
CPU-U		0.99	1.151	1.08	1.065	1.014
% System mode	min	0.35	0.3	1.08	0.55	0.4
	avg	0.41	0.5	4.54	2.39	0.8
	max	0.66	1.0	20.12	12.69	2.7
CPI-K		1.67	1.95	2.24	2.24	2.89
% Load instructions		38%	56%	35%	36%	38%
% Store instructions		15%	13%	20%	17%	28%
Instructions / branch		53.2	14.4	5.4	6.6	10.3
% Mispredicted branches	min	0.3%	38%	15%	6%	5%
	avg	11%	44%	16%	8%	8%
	max	31%	100%	21%	22%	14%
Cache miss cycles (M)		0.217	474	5.937	0.751	1.2
Memory stall cycles (M)		0.6	271	2.9	1.23	1.3
Multicycle ops		0.52	0	5.29	5.11	5.1
MCPI-U		0.0312	0.173	0.03	0.058	0.03
MCPI-K		0.207	0.173	0.297	0.11	0.09
Instruction / invocation	min	149	610,605	5,125	1,587	3,252
	avg	20,885	75,536,952	9,147	3,950	8,126
	max	124,156	420,166,975	19,890	8,758	15,445
Cache misses / instruction		0.0006	0.0182	0.0048	0.0027	0.0015
Wasted cycles (M)		3.51	279.6	19.2	3.197	4.8
% Pipeline stall waste		13.6%	0.05%	28.8%	23.6%	19.5%
Page faults		10	113	61	0	0
System calls		1	7,672,980	19	1	1
Hardware interrupts		36	2969	163	24	41