Lighthouse: Hardware Support for Enforcing Information Flow Control on ManyCore Systems

Sarah Bird
University of California at Berkeley
slbird@eecs.berkeley.edu

David McGrogan
University of California at Berkeley
dpmcgrog@eecs.berkeley.edu

ABSTRACT
We present Lighthouse, a novel multiprocessor architecture based on restricting the flow of information in a shared memory environment. Lighthouse assigns labels corresponding to sets of taint levels to all threads, data, and other system objects, pairs of which are compared to determine the ways in which information may flow between the two corresponding objects. Lighthouse provides hardware acceleration of many common label-interaction primitives, preventing some of the performance loss that has historically accompanied information flow control systems. The use of Lighthouse's features in an actual OS would prevent many common IT problems and supply new possibilities for debugging and other applications.

Categories and Subject Descriptors
D.4.6 [**Operating Systems**] Security and Protection - *Information Flow Controls*

General Terms
Performance, Design, Security, Verification.

Keywords
Information Flow Control, Memory Protection, Manycore, Hardware tagging, Security

1. INTRODUCTION

1.1 Security Issues
Many notorious issues in modern computing – viruses, data theft, and more – stem from vulnerabilities existing in software. These vulnerabilities are distributed over many millions of lines of code. Even if we were to restrict our concerns to a single program, ensuring that any nontrivial piece of software is free of potential security violations is very expensive in money and manpower, assuming it is possible at all; many programs and operating systems rely on third-party drivers or plug-ins to implement some functions, and these cannot be guaranteed safe by the primary program. Additionally, many of the programs available on the Internet have essentially no credentials whatsoever, and it is up to the user to decide whether to risk using them. Rather than attempting to verify the safety of every line of code manually, some projects seek to isolate all security-critical aspects from the remainder of the system, enabling security to be verified more easily if still manually. However, standard operating systems are not equipped to place insurmountable limits on the flow of data between processes or to the outside world.

1.2 Parallel
Projects have been undertaken to create operating systems capable of enforcing the required data-flow isolation. Several such operating systems have indeed been created, but their need to manage permissions for all data using software-only mechanisms causes them to suffer a significant hit to performance. In order for a data-flow restricting system to maintain the level of performance users have come to expect, hardware acceleration of the security functions must be supplied. An additional condition on the nature of such a system is the ascendance of multiprocessing as the dominant hardware paradigm; the new hardware must be designed to support many processors at once. Up to this point all work in data-flow acceleration has been in a uniprocessor context. We have designed a shared-memory architecture that includes all functionality necessary for a hardware-accelerated secure OS.

1.3 Paper Overview
This paper will discuss the relevant work done by others in section 2, explain the goals we set for our design in section 3, lay out the sections and operation of the actual Lighthouse architecture in section 4, convey the results of our experimental and theoretical evaluation of Lighthouse in section 5, present a variety of alternate uses for the labeling architecture in section 6, mention some limitations of Lighthouse's structure in section 7, speculate about potential future efforts and improvements in section 8, and finally give our conclusions in section 9.

2. RELATED WORK
To address the security issues described in the introduction, there have generally been two different approaches used by prior research: information flow control/taint tracking or

fine-grained memory protection. Our system is an attempt to merge the two areas and create a hybrid system, which capitalizes on the advantages of each.

2.1 Information Flow Control

One of the most promising means of implementing the strict principle of least privilege required for ideal security is information flow control. This concept is based on preventing information from reaching any location it is not intended to reach, and generally involves high-level segmentation of programs into functional subunits, rather like a block diagram. These subunits are then allowed to communicate with each other and with the computer's peripherals only in restricted, well-defined ways, preventing information from being leaked, memory from being improperly overwritten, and any number of other malfeasances. Previous efforts in data-flow-restricting OS construction include Asbestos [10], which pioneered the specific type of labeling mechanism we use, and HiStar [9], which extended Asbestos labeling to files and devices. Loki [5] is an implementation of HiStar that uses some hardware acceleration, but experiences no benefit beyond a reduction in the size of the code base that must be trusted as secure. Raksha [7] implements taint tracking in hardware.

2.2 Fine-Grained Memory Protection

Fine-grained memory protection seeks to extend the page-level memory protection provided by operating systems to the level of words or bytes. Sometimes the ability to share regions of memory among processes to increase performance by avoiding data copying is also provided. Although not concerned with security to the same degree as Asbestos and its ilk, they share a need to store permissions for large amounts of data. Our efforts were partially informed by Mondrian Memory Protection [3,4] and Legba,[1] both of which exercise custom hardware to provide protection with performance.

3. DESIGN GOALS

After analyzing the advantages and disadvantages of the current systems and research designs such as Mondrian and HiStar, we created several goals for our design. We sought to take the best features of both information flow control and fine-grained memory protection in order to create a system, which is both flexible and powerful.

3.1 Reduce the Trusted Code

Given the recent security problems with operating systems and the difficulty of statically verifying code, it is important to have a small trusted code base. The smaller code bases can be more carefully coded and rigorously verified. To help reduce the amount of trusted code in our design, we implement information follow control by enforcing protection rules in hardware. Hardware implementation not only provides better performance and more foolproof security, but greatly reduces the trusted code base by reducing a significant amount of software framework to hardware primitives.

3.2 Provide Many Levels of Protection

One of the drawbacks of many current systems is that they only provide a small number of protection rings. Although more levels are sometimes provided many systems only use two of the levels - user and supervisor - which significantly limits the options available for implementing security protocols on these systems. The label and category system creates a far greater degree of granularity in security than merely "privileged" and "unprivileged". By providing more possibilities, new types and applications of security systems are made feasible.

3.3 Low Overhead

3.3.1 Performance

Traditionally hardware is evaluated by performance. This is often the case because other metrics such as programmability and security are very difficult to quantify. Although security is becoming an increasingly issue, people still typically upgrade their hardware systems mostly for the improved performance. While there are definitely some applications for which security is a major concern, most users will not be impressed if their new system is slower even if it is substantially more security. For this reason, one of our major goals was to provide an information flow control system with low performance overhead for typical applications. We attempt to mitigate the cost of the system by putting the common functions such as protection enforcement in hardware. We also try to incorporate much of our design with existing delays in the system so that most of the new delays are overlapped by normal operation of the original system.

3.3.2 Area

As we continue to scale down chips in CMOS technology, each new design has more available transistors. However, despite the availability of many more transistors we think it is important that the design have a low area overhead for several reasons. Although there are many more transistor available, given the increased leakage power with each new process generation, we may not be able to power on all of them at once without overheating the chip. Also, many of the new transistors are going to be allocated to more cores, larger networks and specialized hardware accelerators to help application performance. To keep the area overhead low, we implement the common functions in hardware but use software to handle the more complex and less frequent operations.

3.4 Flexible System

While a security system may perform well on the applications for which it was developed, applications and systems are continuously evolving. The constant influx of new applications and new security threats from the web may require updates to the security system. For this reason, we think it is important the security policy be flexible so that is can be updated when necessary. To this end, we try to separate security policy from protection enforcement. The security policy is implemented in software so that it can easily be changed to suit the needs of different systems, users, or applications. The protection enforcement is done in hardware in order to reduce the trusted code and provide a fast mechanism to support information flow control.

4. DESIGN

4.1 Architecture

This section describes the high level design of our system and different approaches we considered for handling labels. It also describes the division between the hardware and software components.

4.1.1 Labels

Our design uses the labeling scheme designed for Asbestos and HiStar to implement information flow control. The labeling scheme assigns labels to all system objects - threads, memory sections, devices, and so on. These labels describe a level of taint in an arbitrary number of protection categories. Categories are defined by a 61-bit identifier created by a block encryption to avoid data leakage. Taint levels are in the range {*,0,1,2,3} and drive the entire security system; a process can only receive data if the taint levels of all the categories in its label are greater than or equal to the taint levels of the categories in the data's label. In other words, the process must be at least as tainted as the data in every way that the data's label covers. A thread can make itself more tainted in order to read tainted data, but only up to a limit called its clearance, which defaults to 2. The only exception to these rules is the special * ('star') taint level, which is possessed only by the initial creating process of a category and those processes to which the creator gives it. The * level gives untainting privileges, allowing that process to receive and transmit data regardless of the other object's taint level in that category.

In the most common case, taint levels are used to duplicate the standard Unix file permissions. Each user's data has a pair of categories to define read and write privileges. To prevent other users from reading and writing the data, the read taint will be set to 3 and the write taint will be set to 0. As the default taint for objects is 1, they will be unable to write to the data because 0 < 1. Because the clearance for threads is 2, the data's read taint of 3 will not be reachable without permission, and the data will not be readable. Only threads which have had their clearance raised on that user's read taint will be able to read the user's data, and then that taint will prevent them from exporting the data to other objects without the intervention of threads with the * taint level, which is controlled by the user. The total effect is to tightly restrict the flow of information, such that it cannot be transmitted to unauthorized objects.

4.1.2 Relabeling

The flexibility of the HiStar/Asbestos labels creates several difficulties when attempting to implement them in hardware. One of the most difficult issues is that there can be an arbitrary number of categories associated with any label and categories are quite large (61 Bits). This can lead to large, variable-sized protection tables in memory, which require more complex storage methods and hardware to access. Also, if the labels are transported around with the data, then all systems must be designed to handle their variable size, including the network controller and the logic in the processor pipeline. Furthermore, for a small protection granularity the ratio of metadata to actual data would become quite large. Although HiStar/Asbestos labels can make the hardware more complex, we did not want to use a simpler scheme because of our goals of providing many levels of protection and having a flexible design. In order to use the HiStar/Absestos label paradigm without having to deal with variable size throughout the system, we implement a relabeling scheme, which transforms the labels into fixed size tags. For Lighthouse we considered two different approaches to relabeling: a local and a global scheme.

4.1.2.1 Local Relabeling

As discussed in our introduction, parallel computing is seen as the future of computing and all new architectures being developed are multicore or manycore designs. The local relabeling scheme that we evaluated takes advantage of the idea that a system will be composed of many nodes running unrelated processes or applications. A node can be one processor or a cluster of processors in this case. Each node has its own relabeling engine in the network interface, which translates the label into a bit vector of categories. This design relies on the assumption that there is a finite (and small) number of categories active on a node at anytime. The engine keeps track of the active categories and the order of these categories in the bit vector. Bit vectors are stored for the currently running threads and all of the data in the cache. This makes protection checks quite simple because we only need to do a bitwise and on the thread and data bit vectors to make sure the thread vector contains all of the bits in the data vector. However, once the bit vector is full, if a new category becomes active then the scheme gets to be more complex. If another category is now inactive then the new category can replace it in the bit vector. However, this still requires updating all of the bit vectors in the cache and thread vector. Also, any data in the cache that uses the old category must be flushed.

This can create serious thrashing problems if the category being replaced is still active. One approach to handling this problem is to allow software to handle the case where there are more active categories than bit vector space. When a recently replaced category is reactivated the system can use a software handler to compress the bit vector by having each bit represent a combination of categories. This allows the bit vectors to remain the same size and the protection check still uses the same logic. The disadvantage of the approach is that the software handler will need to be run every time a new category is added which may greatly slow down performance if the system has a larger and rapidly changing working set of categories.

Another drawback of the local relabeling scheme is that data is sent across the network with its full-sized labels, which can greatly increase network traffic. Also, the network controllers must be designed to deal with the varied label sizes. Furthermore, because the relabeling is done on a per node basis in order for nodes to communicate the sending node must convert the bit vectors back into labels and then receiving node must then reconvert the labels into its own local bit vector.

4.1.2.2 Global Relabeling

Since new architectures have many simple cores on a single chip and applications may be parallelized and spread across lots of these cores, another approach for relabeling is to optimize for communication between cores. The global relabeling scheme that we considered is designed to have a single relabeling system, which creates tags that can be understood globally. This allows nodes to communicate without having to convert back and forth between bit vectors and labels. In this scheme there is a relabeling engine located with the memory controller that creates a 16-bit tag for each unique label. The engine keeps track of which categories are contained in a given tag. When two tags need to be compared to determine access permissions a software handler is invoked which gets the categories from the engine to compare the labels. The result of this comparison is stored by the relabeling engine in a large table (the Label Comparison Store) containing the two tags and the permissions for read, write, and execute. Near the instruction pipeline, there is a Protection Lookaside Buffer (PLB), which acts as a cache for these label comparisons. The protection check looks up the two tags in the PLB and then checks the permission bits. If the label comparison of the thread and data tags is not present in the PLB, a PLB miss occurs and the value must be fetched from the Label Comparison Store.

This approach does not require translation for different cores to communicate and has a fixed network overhead of 16-bits per data elements. (The size of a data element varies based on the protection granularity chosen.) One drawback of global relabeling is that good performance is dependent on the code's working set of unique labels fitting in the PLB. However, we expect that typical applications should work on data with a very small number of different labels.

The tag size of 16 bits was chosen because it allows a larger number of unique labels for the system, which may be quite important for manycore designs running multiple apps at once and because it is a standard size which may be easier for the network controller, cache and other hardware to deal with.

4.1.3 Software Handler

In order to meet our goal of implementing a flexible security policy in software, we use a software handler to perform the label comparisons. Since invoking a software handler to read the labels stored by the relabeling engine and compare them can be quite expensive, this operation is only performed the first time two tags are compared. When there is a miss in the PLB and the label comparison cannot be found in the Label Comparison Store, then the handler is invoked on the core making the request. Once the label comparison is computed it is stored in the Label Comparison Store. This new entry is visible globally so other cores using the same pair of tags do not need to waste resources in recomputing the comparison. We have chosen to do comparisons reactively instead of proactively because it seems likely that most processes will only operate on their own data which means that many different labels will never interact and computing comparisons for these labels would waste power and computational resources. However, this may add startup cost to applications since many of the initial accesses will require the software handler. Fortunately, unless the labels for the data that an application is accessing are constantly changing, the system should reach a steady state where most of the label comparisons have been computed and the software handler rarely needs to be invoked.

4.2 Microarchitecture

This section describes the lower level design details for the hardware structures that we add to implement Lighthouse. Figure 1 shows the design of the hardware features and the following subsections describe each of them in detail.

4.2.1 Relabeling Engine

The relabeling engine is a controller system. In our design we combined it with the memory controller; however it could be viewed as a separate system. Traditional reads are handled by the memory controller. However, the relabeling engine walks the protection memory table to return the tag with the data. The relabeling engine also handles access requests to the Label Comparison Store. It also keeps track

of the label associated with each tag that it has allocated, handles tag looks up for labels, and allocates new tags for labels which it have not been previously assigned a tag. The controller could be implemented in logic or with software and a generic processing unit.

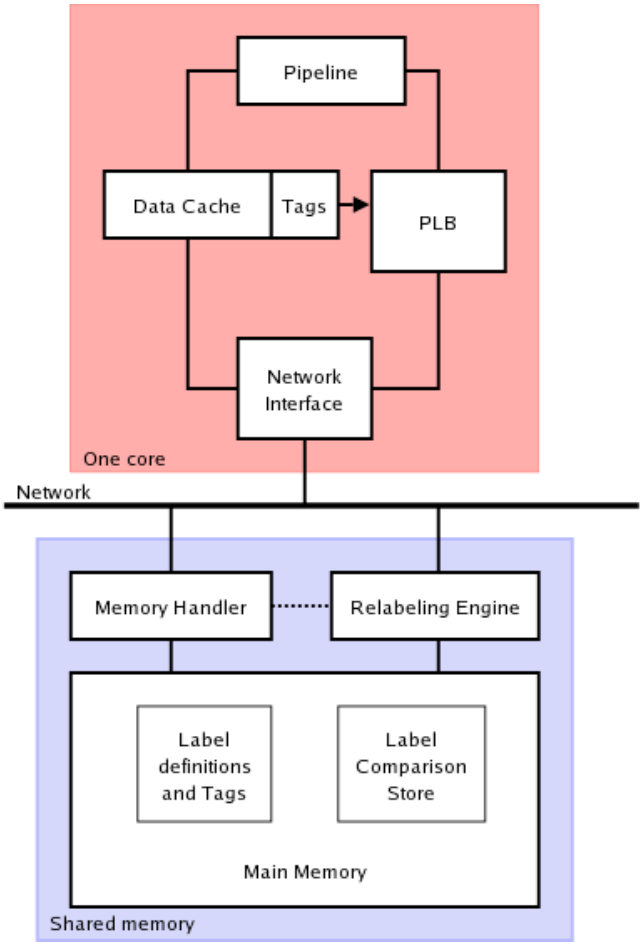

Figure 1. Lighthouse Hardware

4.2.2 Memory Protection Table

The means in which the protection information for main memory is stored has a great impact on the storage overhead of Lighthouse. During its development we considered two primary paradigms for storing this information in a compact way, both of them informed by work done on Mondrian Memory Protection. The first method is based on a series of records containing only the address at which their described region began and the label to be applied to it; the length of the region is calculated using the address held by the next record. This approach provides the maximum flexibility in specifying protection region since they can be of any size. It is an efficient representation of the protection since more than one entry will never be needed to specify continuous regions with the same label. However, a binary search is needed to look up entries in the table. Also, when an entry needs to be inserted in the table all of the other entries must be shifted down. The second method is a design similar to page tables, using a hierarchy of records, which grow deeper with increasing granularity. This approach makes it easier to access entries since it only requires at most as many lookups at the table is levels deep. It lacks the flexibility of the previous approach for specifying regions of varying size since it can only represent regions on the hierarchical granularity of the system. However adding an entry is trivial since the structure is already predefined to allow entry for every data element. Both methods provided the potential for protection granularity down to the word level, but differed significantly in their structure. We judged that the hierarchical method would be more efficient given the more consistent access time and ease of adding entries. However, in the future it may be worth investigating using a hybrid approach of the two.

We have chosen to use a cache-line granularity rather than a word level granularity since it still provides a large amount of flexibility but greatly reduces the overhead in the cache. Assuming a 128-byt cache line, we implemented it as a three level page table in a 32-bit address space where the first level is indexed by the 10 most significant bits, the second level is indexed by the 9 next bits and the last level is indexed by the next 6 bits. This means if the table is fully specified the area overhead in the memory is 285245440 bits/2^32 bits which is 6.641%. The performance overhead is 3 extra memory accesses to walk the table per cache line.

4.2.3 Label Comparison Store

Each label comparison requires software to step in to determine the permission relation between two objects given their label tags - whether one is allowed to receive information from the other. This is an expensive proposition, made potentially more expensive by the effectively limitless number of categories that may exist in either label. To reduce the number of label comparisons as much as possible, the results of each one are stored in main memory in a Label Comparison Store. This acts as a cache of most comparison results that are likely to be requested, and due to the small size of a pair of label tags and the three necessary permission bits to specify result of the comparison the is unlikely to occupy a significant amount of memory space. If desired, an upper bound may be set on its size, but this is unlikely to be necessary except in extremely pathological situations. Our approach uses a direct-mapped vector for the first tag, which is a pointer to linked-list of comparison results with different threads. The entire vector does not need to be specified until of the tags have been allocated so we only allocate the parts of the vector for which tags are in use. The linked-list does not

need to store one comparison per link but could point to a vector of comparisons for the case that most tags have more than one comparison. The logic in the relabeling engine handles comparison lookups and traverses the linked-list.

4.2.4 Protection Lookaside Buffer and Pipeline

This section describes hardware added to the core of the processor, which includes the Protection Lookaside Buffer and the protection check hardware.

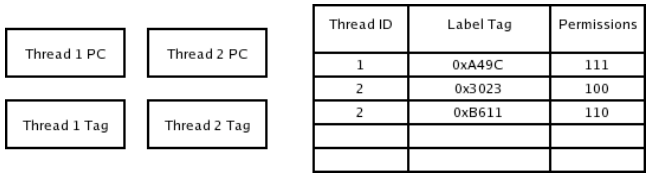

Figure 2. Protection Lookaside Buffer

4.2.4.1 Protection Lookaside Buffer

A Protection Lookaside Buffer (named in analogy to the Translation Lookaside Buffer found in modern processors) is present in each core to accelerate operations on cached data. The PLB is shown in Figure 2. The PLB is a fully associative cache; each line contains the number of the thread and the tag with which the thread tag is compared and the permissions. (Depending on the nature of the programs being run, a fully associative cache may not be the best solution; if there are n possible simultaneous threads with little permissions overlap, an n-way associative cache or n fully associative caches of 1/n size may provide superior efficiency.)

When a label comparison is requested by a thread, the result is stored in the PLB to enable future operations involving data with the same tag to proceed without any stalling of the pipeline. That is, each thread simultaneously held within a multithreaded processor has, in addition to a program counter, register set, and so on, a number of entries within the PLB associated with it. When a thread switches out, all of its entries in the PLB are invalidated to prevent the incoming thread from using any of the departing thread's permissions. Also, if we did not want to invalidate data when a new thread we could store the thread tag in the cache instead of the thread id number. In that case if a thread changed its label or was switched out other threads in the PLB would be able to use the cached permissions if they had the same tag. However, without real applications it is difficult to know whether this is a common case.

4.2.4.2 Pipeline

Figure 3 shows the processor pipeline. Protection checks are performed in parallel with the execution of the instruction. The thread id and the data tag from the cache are used to check for an entry in the PLB.

If the entry is there then the permissions are checked to make sure that the location the instruction does not violate the access rules. If the permission check passes than the instruction commits as in normal operation. If the permission check fails than system issues and exception and the exception handler can decide if the thread should change its label to access the data or abort. In the event that the thread decides to changes its label than it issues an instruction to the relabeling engine with its current tag and the category that needs changing and the new level of that category. The relabeling engine returns a tag for the thread corresponding to the new label. All entries based on that threads previous tag are invalidated.

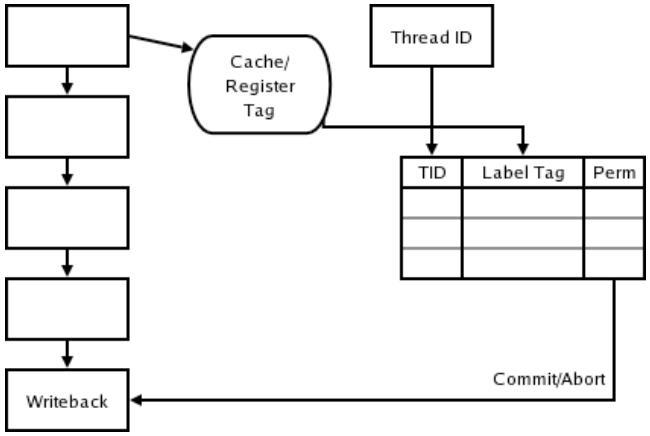

Figure 3. Protection Check Hardware with Pipeline

If the appropriate entry is not in the PLB than a PLB miss occurs. The thread sends its tag along with the tag to compare against to the relabeling engine. The relabeling engine checks the Label Comparison Store for the comparison and invokes the software handler if it cannot be found.

4.2.5 Cache Alterations

Each processor's cache naturally has to preserve the labels of the data within it. We decided to limit granularity to the size and boundaries of potential cache lines so that each line would need space for only a single label tag. We considered the use of multiple tags per line, but at 16 bits per tag the overhead was judged too costly, especially given that there would be a large amount of duplication among the tags due to the likely presence of many memory domains at least as large as a cache line.

If word granularity was extremely important than each word could have a single tag. However, this requires a lot more space in the cache and individual protection lookups for each word in a cache line. It also increases the amount of metadata sent across the network per cache line access.

5. EVALUATION

After reasoning about the design and its overhead, we used a simulator system and models of the expected delays to evaluate the performance overhead imposed by our security hardware. Our evaluation methodology and results are described in the following subsections.

5.1 Methodology

To attempt to quantify the overhead created by our design we simulate the design along with a traditional system with the same architectural parameters and operating system without the added delays of the Lighthouse hardware. We also run our benchmarks on HiStar running on the same architecture to compare our performance with HiStar.

5.1.1 Simulation Infrastructure

The operation of the Lighthouse architecture was simulated using the SimICS system simulator, modified to model the timing effects of the added hardware. The baseline architecture was 20 MHz a Pentium 4 processor with 256 MB of memory running Linux kernel 2.6.15. (The HiStar simulation uses the same architecture but with the HiStar OS). We assumed that the protection label comparison buffer in main memory would be sufficient to hold the results of all comparisons made, thus each comparison would incur the expense of being computed only once; subsequent comparison times would be equivalent to a few accesses of main memory. A custom memory hierarchy was thus created, with a significant penalty for main memory access and a very large cache just above main memory with a moderate access penalty. Simulations were run on a machine with 2 Dual-Core AMD Opteron 2214 processors running at 2.2GHz.

5.1.2 Benchmarks

There are not many applications available, which use information flow control with labels. For this reason we use synthetic benchmarks. We do not include the initialization of the application or the operating system in our benchmark results. We record cycles for the main body of the benchmarks only. In order to provide a simple yet uniform benchmark usable on all systems and exploring memory speed and cache capacity, we used a simple program that repeatedly increments each entry in a vector of integers for the single core. By varying the length of the vector and the number of increment repetitions, the relative behavior of various operating systems in various types of memory loadings could be ascertained.

For the multicore simulation we use 2 different benchmarks, which present basic modes of parallel computation. The benchmarks are described in the following subsections.

5.1.2.1 Stencil

The stencil benchmark implements a 128x128x128 structured grid and performs a stencil computation. This is often used in climate modeling simulations. The benchmark uses coarse blocking and partitions the grid in the x direction for each core.

5.1.2.2 Image Histogram

The image histogram benchmark takes an incoming stream of pixels for a 1024x1024 pixel image and creates a histogram for the red, green and blue values. It uses a map reduce framework to partition the data set and one core combines the histograms from each core during the reduction. This benchmark has little data reuse and essentially represents a streaming application.

5.2 Results

5.2.1 Single Core

We ran the same code on three different simulated systems: standard Fedora Core 5 (FC5), HiStar, and FC5 using our timing model for Lighthouse. Our results are displayed in Figure 4. Small tasks consisted of a vector of 8192 integers being incremented 100 times, medium tasks were a vector of 16384 integers being incremented 200 times, and large tasks were a vector of 16384 integers being incremented 400 times. The results clearly show the overhead incurred by information flow control; the stock FC5 system was significantly faster than both Lighthouse and HiStar. The two secure systems were more evenly matched; although HiStar performed better with the small task, Lighthouse got faster as the tasks got bigger. We attribute this to Lighthouse's caching of label comparison results, something HiStar does not explicitly do.

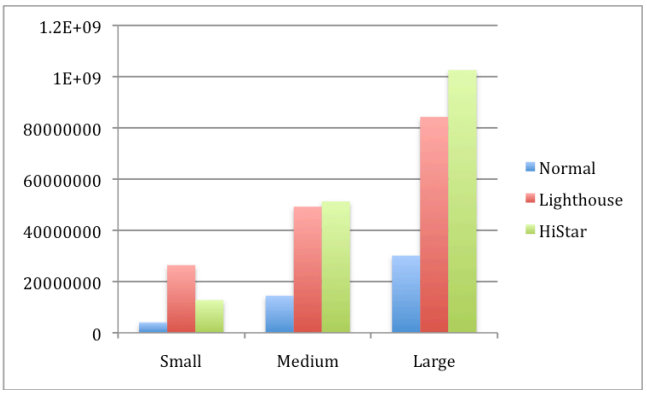

Figure 4. Single core experiments comparing the runtime of a plain system without information flow control to Lighthouse and HiStar using synthetic vector benchmarks. The y axis is cycles.

5.2.2 *MultiCore*

We ran the stencil and map reduce code on our normal and Lighthouse system to compare the performance overhead of our added hardware. Figure 5 shows the results for image histogram benchmark. For the single core and the dual core cases the performance overhead is insignificant. However in the case of the quad core system, the added delay cycles from the software handler interact poorly with the barrier causing the synchronization time to increase significantly. This may not be as noticeable in a real application because the initialization time period will be dominated by the body, which should not need to invoke the software handler.

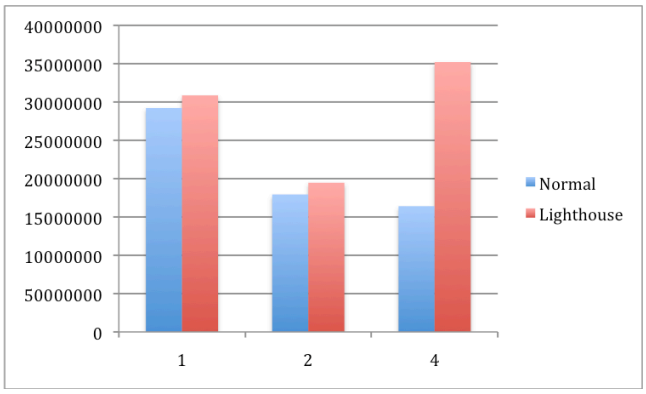

Figure 5. MultiCore experiment comparing the runtime a plain system without information flow control to Lighthouse using the image histogram benchmark. The y axis is cycles and the x axis is processors.

Figure 6 shows the results fot the stencil benchmark. The performance overhead for our Lighthouse system is around 20 to 30 percent for single and dual core systems. The quad core systems suffers from the same synchronization problem as the quad core in the image histogram code.

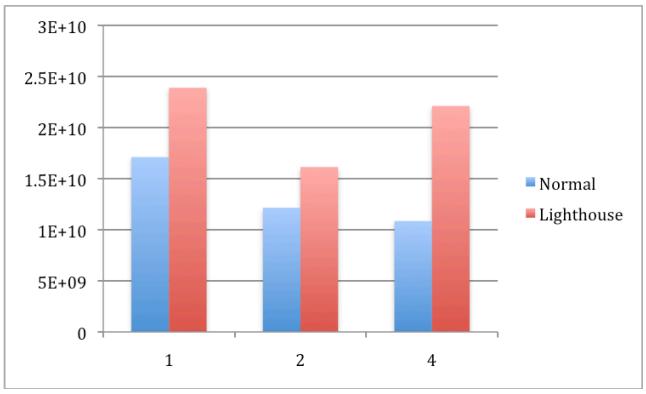

Figure 6. MultiCore experiment comparing the runtime a plain system without information flow control to Lighthouse using the stencil benchmark The y axis is cycles and the x axis is processors.

5.3 Interpretation

5.3.1 *Lack of Real Applications*

Unfortunately, we cannot claim from these results a conclusive decision for Lighthouse or HiStar. The timing model is limited in scope, and while it does significantly delay the first access to any given memory location it is unable to account for accesses to locations belonging to the same process - that is, having the same label and thus not needing another long calculation of a label comparison. Additionally, the delay may not accurately portray the amount of time needed to do the label comparison. A more complicated model, capable of storing information on which program objects have been accessed, could provide more accurate results, but we were unable to create such a model in the time allotted. Furthermore, no present applications are written to take advantage of the Lighthouse architecture. This prevents any meaningful comparison in the performance of consumer-type applications, and even standard benchmarking algorithms can only provide limited information because performance alone is not the objective. Security is also important, but is impossible to quantify.

5.3.2 *Theoretical Performance*

We can, however, state with certainty the conditions under which Lighthouse will excel or fail. As noted above, Lighthouse requires a significant amount of time to compute the available legal interactions when two label tags are compared for the first time. If a program frequently creates new categories, changes its taint level, or tries to read memory from many different protection domains, it will invoke the software-based label handler many times, creating a great deal of lag in the program. On the other hand, if a thread sets up only a few categories and operates within them exclusively, the information regarding the thread's interaction with the memory sections will be computed once and then cached in the core, never again to incur any additional delay to the thread. This latter behavior is what we expect from most well-behaved programs, and we therefore predict that Lighthouse will fare well in most environments.

6. OTHER USES

The label-manipulation capabilities of Lighthouse have many potential applications beyond basic operating system security. For example, the ability to place restrictions on what code may read and write what memory is a fantastic tool for debugging. This was suggested during the development of Mondrix [4], when the authors found bugs in kernel code that was heavily tested and commonly used. By integrating label manipulation into debugging compilation, any number of possibilities might be realized.

Along similar lines, a major issue with developing applications for the online environment is the potential

malice of the end user; much work has gone into avoiding such attacks as SQL injection. One idea is the 'tainting' of all data that comes from the user, such that it can never be executed. This is obviously an excellent fit for Lighthouse, which works with tainting natively. More detailed tainting systems are also possible, keeping track of the data from individual users such that users might never see each others' data.

In an extreme case of the above, governments are obviously very concerned with the protection of their information. Computer systems with security verification built into the hardware would be a great asset to governmental systems administrators, as sensitive information could be prevented in hardware from reaching any potentially compromising device - the network connection, for example.

Finally, the use of tags is not limited to tainting. As shown in the Shift-M project [2], tags can be used in message passing frameworks, to place permissions on messages or verify them.

7. LIMITATIONS

It is possible for a malicious process to continually create new categories and label comparison requests, thus bogging down the label comparison system. Programs which behave this way normally, such as a webserver with many unique users that share some information, will also experience significant slowdown. We have assumed that both category creation and label comparison will be infrequent; the addition of a request buffer that will accept a maximum of one request per hardware thread at a time should prevent total denial of service, but is hardly a perfect solution.

8. FUTURE WORK

Many possibilities exist for future elaboration of the work presented in this paper. Foremost among these is the construction of a more accurate simulation model, enabling a truer comparison between the performance of Lighthouse and the performance of other architectures; our current model is a very rough approximation of Lighthouse's timing. Improved models could keep track of the actual labels in circulation, providing much better accuracy. Similarly, Lighthouse could be implemented in actual hardware via FPGA, as was done by other security projects, including Mondrian and Loki. Some processors, including the SPARC, have versions available for academic modification, which would be well used in such a project. Finally, the authors of HiStar and Loki were quite concerned with the size of the trusted code base, even above performance. Presently the reduction in trusted code permitted by the Lighthouse hardware is unknown, which makes it impossible to compare on that front. Actual implementation of the necessary kernel software would reveal any improvements, and additionally allow it to be actually used for real-world purposes, such as the aforementioned debugging tools, and potential additional research and modifications.

9. CONCLUSIONS

Lighthouse is a means for enforcing information flow control within a shared memory multiprocessor architecture that requires relatively minor modifications to the cores and an added label comparison engine in the memory manager and software. It provides an implementation of Asbestos-style labels, preventing even the system administrator from viewing the data of other users. It shows potential to outperform software-only implementations of information flow control by placing most of the functionality in hardware, and will likely have a smaller trusted code base than other hardware-assisted implementations. By adopting information flow control, a variety of problems that plague modern computing may be dealt with, including many viruses and vulnerabilities. Existing architectures provide little substantial support for flow control, whereas Lighthouse enables it to be accelerated with hardware support.

10. ACKNOWLEDGMENTS

Our thanks to John Kubiatowiz and Krste Asanovic for giving advice and guidance. Also Nickolai Zeldovich for providing information and answering questions about HiStar.

11. REFERENCES

[1] Adam Wiggins, Simon Winwood, Harvey Tuch and Gernot Hesier, Legba: Fast Hardware Support for Fine-Grained Protection.

[2] Colleen Lewis and Cynthia Sturton. SHIFT+M: Software-Hardware Information Flow Tracking on Multicore. University of California at Berkeley, May 2008.

[3] Emmett Witchel, Josh Cates, and Krste Asanović, "Mondrian Memory Protection", Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X) , San Jose, CA, October 2002.

[4] Emmett Witchel, Junghwan Rhee, Krste Asanović, "Mondrix: Memory Isolation for Linux using Mondriaan Memory Protection", 20th ACM Symposium on Operating Systems Principles (SOSP-20) Brighton, UK, October 2005.Tavel, P. 2007 Modeling and Simulation Design. AK Peters Ltd.

[5] Hari Kannan, Nickolai Zeldovich, Michael Dalton, Christos Kozyrakis. Architectural Support for Minimizing Trusted Code.

[6] Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Michelle Osborne, Steve VanDeBogart and David Ziegler. Make Least Privilege a Right (Not a Privilege). Proceedings of the 10th Workshop on Hot Topics in Operating Systems, Santa Fe, NM, June 2005.

[7] Michael Dalton, Hari Kannan, Christos Kozyrakis, Raksha: A Flexible Information Flow Architecture for Software Security. Proceedings of the 34th Intl. Symposium on Computer Architecture (ISCA), San Diego, CA, June 2007.

[8] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing Distributed Systems with Information Flow Control. In Proceedings of the 5th Symposium on Networked Systems Design and Implementation, San Francisco, CA, April 2008

[9] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation, Seattle, WA, November 2006

[10] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek and Robert Morris. Labels and Event Processes in the Asbestos Operating System. Proceedings of the 20th Symposium on Operating Systems Principles, Brighton, United Kingdom, October 2005.