Dynamic Kernel Modification and Extensibility

Arun Kishan
Department of Computer Science
Stanford University
akishan@stanford.edu

Monica Lam
Department of Computer Science
Stanford University
lam@stanford.edu

1. Introduction and Motivations

Today, one finds a plethora of computer programs that are designed to run without interruption. Common examples include financial transaction systems (e.g. the VISA computer system), telephone switches, certain web services, and air traffic control systems. Such systems are typically dubbed "mission critical," suggesting that a temporary faltering in their operation is tantamount to catastrophic failure. Thus, when presented with a bug fix or an upgrade to such systems, the typical paradigm of "halt applications, apply upgrade, restart system" may be deemed unacceptable. Solutions geared towards using redundant fault-tolerant hardware coupled with specialized software are often expensive and complicated by the need to maintain consistency between the various system mirrors. Clearly, for such systems, on-line update presents an ideal solution. With waning end-user patience for operating system reboots, even systems and applications that do not necessarily require on-line update functionality stand to benefit from its use.

An operating system is an example of an application that benefits from dynamic update, but certainly may not require it (depending on the context in which the system is deployed). Nevertheless, the inconvenience of restarting the entire system any time new code needs to be introduced into the kernel has driven operating systems toward a limited notion of dynamic extensibility. In particular, most operating systems feature the ability to load additional code via dynamically linked libraries (DLLs) or kernel extensions (kexts or kmods), which are typically introduced into the kernel in response to some manner of I/O based activity. This form of dynamic extensibility and update is known as "plug-in" extensibility – the manner in which code will be introduced into the system is predetermined. Obvious limitations of such a scheme include the inability to update code once it has been loaded and the inability to evolve loaded code beyond the usage scenarios envisioned by the developer. The key issues in the design of an updateable system then become centered around which components of the system may be updated without programmer anticipation, and the timing with which such an update may be made. In the ideal scenario, we would find that virtually any component of the operating system could be updated, and at any time (i.e., regardless of whether the target code is currently active or inactive). Such a system would be of great utility not only to mission critical applications, but also to kernel developers and researchers (who can dynamically insert and remove instrumentation code without forcing a kernel recompile and reboot), as well as typical end users.

The issues of dynamic kernel extensibility and on-line update and their applicability to commodity operating systems are intrinsically related to the notions of runtime code generation and dynamic software feedback. Once an infrastructure has been developed that supports the former functionality, one can imagine that the latter set of features may be introduced with little difficulty. Indeed, the great success that code synthesis and software feedback have enjoyed in

the implementation of the Synthesis kernel [13, 14, 15] provides ample impetus for the development of a run-time update infrastructure, independent of the benefits described above. Examining the predicted evolution of operating systems, from monolithic, to microkernel-based, and finally to a fine-grained modular design (in which kernels are modularized at the object/abstract data type level), the Synthesis research suggests that the applicability of code generation increases as one progresses between the various operating system designs. In particular, the trend towards finer grained modularity reduces the internal dependencies created by shared data as found in coarse-grained modules, paving the way for dynamic optimization. Microkernel systems, which generally feature server modules running above a minimal kernel, tend to experience an increase in the cost of invoking operating system functions, particularly when a user-level emulation library is used to simulate a monolithic operating system. Thus, although individual system calls are typically implemented efficiently, the end-end latency in such systems tends to be rather high. As such, microkernel systems appear to be reasonable candidates for run time code synthesis and software feedback. Further, such systems tend to feature abstract kernel interfaces, which shield applications from the underlying implementation details. As such, code generation may replace a single routine with various application-specific specializations, potentially yielding large performance gains. In contrast, monolithic kernels tend to provide direct access to kernel data structures, which sharply limits the applicability of dynamic code generation. Finally, as kernel design tends towards greater encapsulation and modularity, the overhead of the layered system design that results (particularly in a system such as Mach, which carefully separates its machine dependent and independent portions) increases. Each layer is often fully encapsulated, obfuscating information that may be pertinent to dynamic optimization. Consequently the performance degradation observed by the end-user may be worse than linear in the number of layers. By collapsing these layers at runtime, one may enjoy the benefits of encapsulated kernel design without suffering the performance penalties. With today's abundant computing resources, it has become feasible to once again consider the use of microkernel based commodity operating systems; examples include Apple's recent Mach-based Darwin operating system. Exploration of dynamic code synthesis and feedback in the commodity microkernel context thus certainly provide fertile ground for research.

Our primary goals in developing the dynamic kernel modification system described in this paper are thus:

- **Unrestricted dynamic update of kernel code** – We would like to dynamically insert or remove code from any portion of the kernel. If possible, we do not want to be subject to the restrictions imposed by other schemes, which stipulate that updates must be introduced at the granularity of entire functions or entire modules.

- **Avoidance of timing constraints** – Unlike most previous systems, we would prefer our system to be impervious to timing considerations. In particular, it should be possible to dynamically inject additional code into any portion of the system regardless of whether the target of modification is currently active. The performance impact of dynamic code injection must also be minimized, or else the usefulness of any dynamic code generator based upon this technology will be compromised.

- **Provide a highly modularized framework for portability** – As discussed in the related work section, researchers have developed and deployed a similar system for the Solaris

operating system. It is unfortunate that the source to this system remains unavailable, preventing a port of this technology. In the design of the current system, each component has been highly modularized along machine and platform independent lines. While certain hardware and platform specific caveats cannot be avoided, it is our hope that the modular design described in this paper will promote our system's portability.

- **Research the applicability of code synthesis and dynamic feedback in the microkernel context** – Though the Synthesis project alluded to the applicability of these technologies to microkernel based operating systems, a great deal of this discussion was limited to speculation based on theoretical results. In developing a dynamic kernel update system for a modern microkernel based operating system, we hope to explore the applicability of Synthesis' key technologies outside of the context in which they were devised. It is our hope that the results we derive will assist the field at large in analyzing the modern day viability of alternative (i.e., non-monolithic) kernel designs.

2. Related Work

2.1 Dynamic Software Updating

Much work has been done in the area of dynamic software updating. Hicks et al., at the University of Pennsylvania [6] have developed an operational software system which allows for runtime installation of code update. Typical code updates include bug fixes, improvements to runtime performance, and new functionality. Such code updates come packaged in a "dynamic patch," which contains an accompanying segment of transitional code designed to migrate the state of the running program to one compatible with the newly introduced code. One may thus define a "dynamic patch" as the pair (c, S), in which c is the new code and S the aforementioned state transformer function. Note that the complexity of the state transformer function varies from trivial to complex, depending on the state of the running program and the nature of the code being introduced into the system.

The principal investigators designed the system largely to meet the following criteria: the system needed to be flexible, allowing for update of nearly any of its components, without requiring the original developer to anticipate such runtime changes; it needed to be robust, automatically verifying the correctness of the patch to be installed; it needed to be easy to use; and finally, its performance impact needed to be negligible. To a large degree most of these concerns have been addressed. Whereas other systems have limited what can be updated by restricting the granularity of update (typically to the function, module, or entire program level) as well as the time of update, here one sees that changes may be affected at the granularity of individual definitions, functions, types, or data with little to no restriction on the time of update. With such increased flexibility or "agility" comes a potentially unavoidable decrease in the enforceable safety of code updates (i.e. the ease with which one could prevent the code update from crashing the system or otherwise causing it to behave improperly) [7]. Through the use of Typed Assembly Language, code patches are made to respect various facets of type safety (e.g. pointer forging is prohibited, as is jumping to arbitrary code or modifying non-local stack data) as found in other languages such as Java. However, the code patches, though typed, are in native assembly code and thus do not suffer from the performance degradation experienced by these other languages. Ease of use is afforded by divorcing the process of dynamic patch generation

from the actual development cycle; patch generation is largely automated and operates at the source level by comparing the new and old versions of the code. Total automation is not possible as it is, in the general case, an undecidable problem. It is up to the developer to fill in any holes in the code update and the associated transformer function that are left behind by this process.

Using native code for the code updates was necessary to minimize the overhead of the dynamic update system. Further control over the performance of the system code was imposed by carefully designing the mechanism by which the patches would actually be introduced into the runtime system. Typical operating systems today provide dynamic linking facilities to introduce new code (on demand) into the address space of running processes, though there is little to no ability to replace definitions that are already present (linked). In the case of code and data updates, two options presented themselves: a variant of dynamic linking known as code re-linking that would actively redirect all references to the old definition to the new definition, or a technique known as reference indirection, which would force all references to indirect via a global table which the code updater could easily modify to refer to new definitions. Note that in both cases, the state transformer component of the dynamic patch would be responsible for updating references in the running program state to refer to the newly introduced definitions -- the task would not be simplified by the choice of patch introduction. Thus, reduced purely to an issue of performance and implementation ease, the code re-linking mechanism was chosen as it removed the universal overhead of pointer de-reference introduced by the alternative method, while its implementation borrowed heavily from existing dynamic linker technology.

The viability of the complete dynamic update system was successfully demonstrated via the incremental on-line development of the Flash-Ed web server. One sees in the experimental results that the performance differential between the dynamically updated software and the statically updated (and statically recompiled) software is in the noise. Measured load-time overhead was exactly as expected – equivalent to the time required to re-link and verify the type safety of the patch. Though such times are far from non-existent, the end-user would typically prefer such a fleeting pause in execution to a complete operating system reboot.

Certainly there is much our current work can derive from the experiences of Hicks, et al. However, in our case, certain unique difficulties present themselves that tend to separate our work. One such difficulty is the issue of update timing. Whilst the dynamic update system allowed for patching of inactive code without incident, several race conditions could arise in the case of active code update [6]. The solution adopted by the system designers was to require dynamically updateable programs to be cognizant of runtime updating, requiring the developer to annotate code locations deemed safe for run time modification. A primary shortcoming of this solution is the mental burden imposed on the developer, who must now take great care in choosing sites safe for runtime update. Choosing such locations can certainly be difficult, particularly in multi-threaded contexts. Unfortunately, one cannot avoid the issue of simultaneously executing threads inside a fully re-entrant symmetric multiprocessing kernel. Additionally, restricting the locations at which additional code may be introduced into the kernel could have a detrimental impact on the types of applications for the technology that we have envisioned. It is also unrealistic to suggest the kernel be re-written in a language that can be compiled to produce type-safe assembly. Finally, though many of the issues present when handling user space programs carry over to kernel space, several additional complexities – be it in terms of implementation, debugging, or the like – will inevitably present themselves.

2.2 The Synthesis Kernel: Runtime Dynamic Optimization Techniques

The notion of run time code generation and dynamic software feedback at the operating system level were previously explored by the Synthesis project [13]. In large part the designers of the Synthesis kernel saw these techniques as a way to address the "end-end latency" problem emerging in contemporary operating systems. For typical multimedia audio/video applications, a pipeline could be traced from the input data entering the system via a device, passing through numerous levels of kernel and application processing, and ultimately being sent to an output device. Adding additional steps to this pipeline in accordance with the growing demands imposed by typical applications tended to adversely affect both the throughput and the end-end latency of the pipeline, impairing its ability to successfully service application demands. One possible solution would have been to reduce the granularity at which the system performed data movement and CPU processing, such that additional pipeline stages added little to end-end latency. Unfortunately, with such small stages, it would not be possible for the operating system to distribute the overhead of context switch, interrupts, and system call invocation across a large volume of useful work. By using code generation, Synthesis aimed to preserve the general pipeline structure but reduce the latency of critical kernel functions [14, 15]. Software feedback techniques aimed to reinforce this technique by controlling variance in the latency introduced by the operating system scheduler. In particular, the software feedback mechanism challenged the standard notion that the processes in the pipeline stages were largely independent of one another and instead viewed them as tightly coupled producers and consumers. As a result, the kernel generated fine-grained process scheduling policies to facilitate the movement of data between processes and promote real-time I/O processing.

Synthesis' code synthesizer generates specializations of kernel routines tailored for specific situations. To accomplish this end, it employs three primary techniques: factoring invariants, collapsing layers, and making data structures executable. "Factoring invariants" performs a transformation much like the constant folding optimization performed by standard compilers. Given a function and a set of parameters, Synthesis generates a specialization of this routine assuming a subset of the parameters is given certain values (i.e., these parameters are taken to be invariant). By propagating the results of this assignment through the routine, certain fragments of code could hopefully be dynamically analyzed (e.g., the result of compare instructions) and entire portions of code simplified and/or eliminated. The obvious benefits would include improved straight-line execution of code (fewer pipeline stalls would be introduced by wrongly predicted branches) and reduced instruction cache pollution. Synthesis dynamically chooses the invariants in order to maximize the benefit of routine specialization; primary benefactors of this technique are routines written to handle a variety of cases, though their dynamic function is determined by passed parameters. "Collapsing layers" examines call chains through numerous layers and opportunistically performs aggressive code inlining, in the hopes of eliminating function call overhead and paving the way for the code synthesizer to perform further optimizations. "Executable data structures," though a novel concept, have much less applicability. One compelling example used in the Synthesis system is in the process scheduler. By merging process save/restore code with the process queue and the hardware timer interrupt, Synthesis is able to nearly eliminate data structure traversal during context switch. Synthetic machines, another concept unique to Synthesis, which present running user programs with a high-level interface to kernel services, also benefit from the Synthesis code synthesizer (the Synthetic machine is a virtualization of the underlying hardware and is functionally similar to a Unix process or a Mach task). Typically, high-level kernel interfaces, though easier to use,

necessitate complex (and often slow) implementations; simple kernel calls such as those found in the V kernel and Mach often feature simple and elegant implementations, but necessitate many layers of application-level code to compensate for their functionality shortcomings. With runtime code synthesis, the kernel is able to optimize the complex code underlying the high-level kernel interface presented to applications.

A surprising result of the Synthesis project was the synergistic effect between the various components of the system. In particular, the software feedback system and code synthesizer tend to reinforce and heighten each other's effectiveness. Indeed, general feedback systems are highly input sensitive in terms of their complexity and cost; they perform as desired only within a constrained input range. For this reason, code synthesis may generate specialized feedback mechanisms on demand, as dictated by the observed input streams. These specialized feedback systems may then respond well to highly volatile situations in which parameters tend to change frequently but nevertheless within the range handled by the system. Code specializations generated by the synthesizer, on the other hand, will not perform well under such situations as they are tailored to static, invariant parameter values. Together, Synthesis demonstrated that the two techniques could combine to lower the execution overhead of the operating system kernel and mold the kernel to its execution environment.

The primary shortcoming of the Synthesis kernel is its confinement to the world of academia. As an academic operating system -- even with emulation support for more popular operating systems such as Unix -- it faces little chance of widespread adoption. Our project hopes to explore the means by which some of the features present in the Synthesis kernel may be retrofitted onto unmodified commodity operating systems. Synthesis left open the research question as to whether or not commodity microkernels, such as Mach, could benefit from its runtime optimization techniques; we hope to explore this question as well.

2.3 Dynamic Kernel Instrumentation

A relatively recent work of great pertinence to our current project is the research into fine-grained kernel instrumentation conducted by Tamches et al [17, 18, 19]. The KernInst system developed therein provides a fully dynamic means by which nearly any machine code instruction in the Solaris operating system may be instrumented with user-specified code (notions of kernel extensibility as found in the VINO or Synthetix operating systems operate only at the function or module granularity, and thus offer considerably less flexibility). The means by which the patch code is generated is orthogonal to the means by which it is ultimately installed into the running Solaris kernel. In the design described in [17, 18], a user space program performs a great majority of the work. Through interaction with a specialized kernel component (which may be dynamically loaded and unloaded) the user space process uses the runtime kernel symbol table to parse the in-core kernel machine code, analyzing basic blocks and gathering register usage information (in order to generate valid code patches). A generated code patch is then placed into memory shared between the user process and the kernel, and the user process signals the kernel component to perform the final code splice (a privileged operation).

Various difficulties that arose in the implementation of KernInst are discussed in [17]. Though at times largely architecture dependent, the problems and their solutions are generally applicable to most processors. As part of the instrumentation process, for example, the instruction at the instrumentation site is relocated to the code patch, and a branch to the code patch is written at the instrumentation address; it is executed immediately before a branch

instruction (or equivalent sequence) designed to return to the kernel code at the address immediately following the patch site. For most instructions, this works as expected; the exceptions arise with PC-relative branches, which compute their target using an offset that is encoded as part of the instruction in conjunction with the current PC. Once relocated to the patch memory, the semantics of such instructions are no longer preserved. Another difficulty that arises (in RISC architectures) is the limitation placed upon the branch displacement. Due to this limitation it is possible that the branch instruction inserted at the instrumentation point will be unable to reach the code patch. Tamches' system addresses this problem by using "springboard" memory near the original instrumentation which can accommodate an (on RISC machines, typically four) instruction sequence to perform a register indirect jump to the code patch (this sort of branch will typically cover all addresses that can be accessed by the processor). These scrap regions are allocated in an ad hoc fashion, typically from code pages that the kernel will never execute again (e.g., boot routines or module initialization routines).

A preliminary application of KernInst demonstrated its usefulness as an investigative tool designed to facilitate kernel and application tuning. Using carefully placed instrumentation, KernInst observed the behavior of various kernel routines as invoked by the Squid web cache software when subjected to an aggressive web proxy benchmark. Care was taken such that hardware counters used to measure time were saved and restored upon context switch (which in turn involved instrumentation of the context switcher) in order to ensure the execution times measured were virtual (CPU), rather than wall, time measurements. Using this technique, the source of the bottleneck was isolated to the blocking file open call made in the application's primary select loop, which in turn spent most of its time in the kernel ufs_create call and the lookuppn call. Upon closer inspection, it was found that the plethora of files used by the Squid caching scheme overwhelmed Solaris' Directory Name Lookup Cache (DNLC), forcing disk reads of directory inodes. Likewise, Squid's scheme of avoiding file deletion in favor of file truncation, with the hope of sparing expensive file meta data updates, appeared to result in expensive "no-op" operations as files were truncated and resized to their original size, all the while incurring the cost of synchronous inode updates. Adjusting a kernel parameter increased the number of DNLC entries, and changing a few lines of cache file management code within the Squid addressed both of these problems, respectively.

Another application of KernInst's dynamic kernel instrumentation technology addresses the observation that seldom executed code (cold code), prevalent in most operating system kernels due to abundant error checking, tends to pollute the hardware instruction cache and degrade performance in the common case. By dynamically gathering control flow graph execution counts, KernInst may strategically reposition blocks of kernel machine code in order to improve straight-line execution performance. Typically, this may involve elimination of some unconditional branches, as well as the placement of the commonly taken case of a conditional branch sequentially after the instruction (this may involve inversion of the branch condition). Enforcing sequential code layout in this manner improves instruction cache utilization as a larger percentage of instructions occupying a cache line will tend to be executed. Instruction cache conflict misses can be reduced if code determined to exhibit temporal locality is appropriately spatially positioned as well. Before KernInst repositions the kernel code, it takes care to ensure all affected instructions are relocatable or can easily be converted to a relocatable form, as their instruction addresses will likely change. Run-time use of this technique showed significant reduction in instruction cache misses and corresponding CPU stalls, corresponding to a notable improvement in performance.

KernInst demonstrated the feasibility of developing a fine-grained, dynamic kernel instrumentation system for a commodity operating system. Having developed and deployed the system for Solaris, a few open questions remain. In particular, it is instructive to explore the feasibility of developing such a system for an alternative host architecture, operating system and/or kernel structure. Additionally, whilst run-time code positioning was performed, little has been said regarding the feasibility of performing dynamic code transformation such as constant propagation or code inlining (much like the optimization performed by the Synthesis code synthesizer). In particular, in most modern operating systems, one finds that the time spent within the operating system has been dramatically reduced; thus, one must carefully gauge the tradeoff between the time spent generating dynamically transformed kernel code and the time saved executing the new code using modern CPUs. It is unfortunate that the KernInst source base has since been closed, preventing further academic exploration of these questions. Certainly, it is our hope that the system we have developed and described herein can assist in addressing some of these questions.

3. Mechanisms

3.1 Overview

In order to provide full flexibility to the client of the proposed infrastructure, it was decided early on that dynamic modification would be conducted at the granularity of individual machine code instructions. By exposing the underlying machine code, all statically performed compiler optimizations would be visible to a dynamic optimizer. Using such knowledge, the run-time optimizer can make informed decisions regarding the sorts of optimizations it would like to try (constant propagation, etc.), as well as the machine registers it would like to use in the generated code. Having thus made the decision to conduct instrumentation at this granularity, our choice of host architecture followed suit. To avoid the complexities afforded by CISC based architectures, we opted to work with a RISC based CPU. Our primary concern was the decoding complexity of CISC instructions as well as CISC's typical non-conformance to uniform length instructions. The former point is only of note insofar as the kernel machine code analysis is concerned; our system has been designed with sufficient modularity, however, that code analyzers for varying architectures may easily be plugged in. Non-uniform length instructions, on the other hand, greatly complicate the means by which our secondary goal (dynamic code introduction extricated from timing constraints) can be realized. Though we discuss the theoretical solution to this problem, our prototype employs a much simpler approach that is possible when considering only fixed width instructions. We have chosen a fairly widely supported and powerful RISC CPU: the Motorola PowerPC chip. Our current efforts have been concentrated upon the MPC750 incarnation, though as no chip specific features have yet been used, they are generally applicable to the entire PowerPC family.

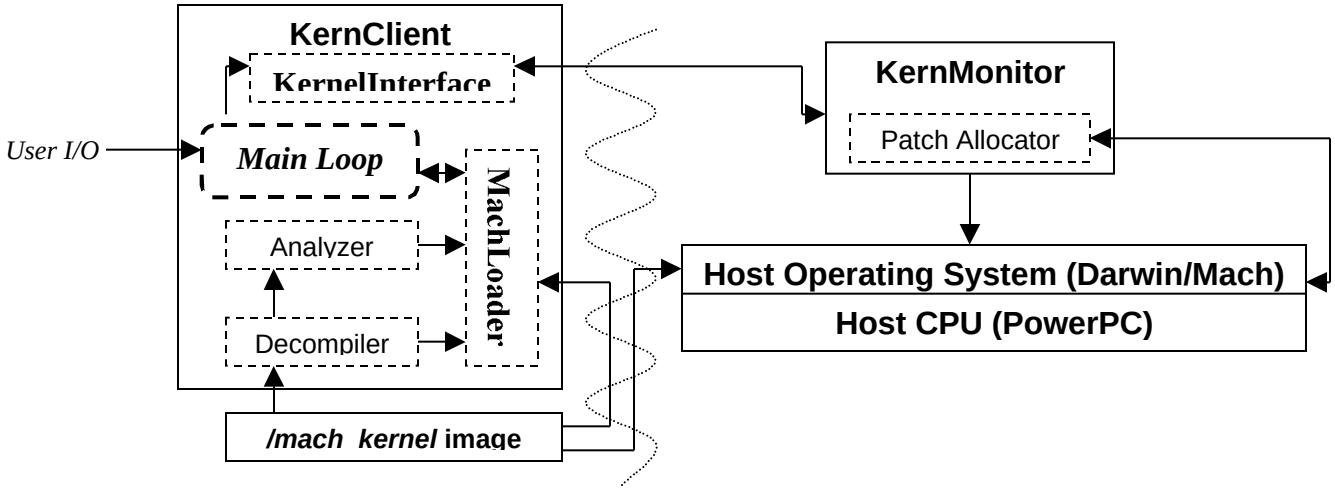
Regarding the choice of host operating system, our choice was largely determined by the research motivations outlined by our fourth goal above. Once our system became functional, we desired to use it to research the viability of employing various run-time optimization techniques in a modularized microkernel-based commodity operating system. Mach, historically an academic operating system known for its less than-desirable industrial performance, has recently been bolstered to mainstream status as the result of a new operating system effort spearheaded by Apple Computer. While the new system has often been lauded for its modular and clean design

since its introduction, it has also been slandered for its mediocre performance when compared with its monolithic counterparts (most notably Linux, though some would posit that the Windows NT/XP line of operating systems may be included under this banner). Thus, any results we would obtain would be of interest to those on either side of the operating system divide. Additional benefits of working with Apple's new operating system include native support for PowerPC hardware in addition to freely available source code for all portions of the kernel (the open source variant is known as "Darwin"). We note that the latter point is purely an issue of convenience; since our analysis operates directly on the kernel binary code, high-level source is not strictly required. Nevertheless, the interpretative overhead of parsing compiler generated and optimized machine code tends to favor readily available high-level source; we have found the availability of such source to be invaluable in the development of our system.

3.1.1 System Architecture

A high level overview of the system architecture is presented in Figure 1. Note that the system has been highly modularized in the hopes of realizing the third goal outlined above. Each component may be tailored to a differing host operating system and differing host CPU, independent of the other components of the system. No particular facility assumed by our system is unique to Darwin or the PowerPC chip (save for the hardware and system idiosyncrasies we have noted below), and thus should pose little difficulty during a port.

Figure 1: System Architecture



One point to note is the clear dichotomy between the user space and kernel components of our system. As far as possible the volume of kernel resident code has been minimized; this may be observed by noting the physical size of the kernel component is roughly a fourth the size of the user space component. The reason for this design is exactly what one would expect: bugs in kernel resident code have the capability to crash the entire system or otherwise affect the system's behavior. Misbehaved user space processes, on the other hand, are simply terminated by the host operating system. Consequently, the responsibilities of the user space component are

greatly increased in terms of the number of tasks it is expected to complete, whereas the responsibilities of the kernel component are reduced to the bare minimum set of privileged operations needed to implement our system. Details of each component are discussed in the remainder of this section.

3.2 User space component

In the above design, it is the responsibility of the user space component to analyze the kernel machine code and decompile it into some intermediate form fit for data flow analysis, generate a call graph of all kernel routines, conduct live register analysis on the resulting control flow graphs, and lastly, accept user input instructing the system to add or remove kernel patches. Successful completion of these tasks requires code that can analyze the native object file format, interpret instructions present in the host CPU instruction set architecture (ISA), perform compiler-style data flow analyses, and communicate with a kernel service. The modularization of KernClient (the user space process) as shown in Figure 1 above roughly reflects this distribution of labor.

3.2.1. MachLoader

A large portion of the user space code is present in the MachLoader module. The primary responsibility of this module is to parse the kernel's machine code into distinct code segments representing each kernel routine. Assuming the invariant that the kernel file used to boot the hardware is a world readable file located in the root directory ("/mach_kernel"), an instance of the MachLoader class opens and parses this file. In this preliminary stage no attention is paid to the actual underlying machine code instructions; rather, by using the static symbol table located in the kernel file, the MachLoader object delimits the virtual addresses specified in the object file into a sequence of code segments. Care is taken to ensure that the code in the static kernel file cannot be relocated, such that the virtual addresses specified in the kernel file correspond to the in-core VM addresses of the machine code. Today, with modern memory management hardware and software, object files are generally loaded into memory without relocation, considerably simplifying our task.

It is also to the MachLoader's benefit that the kernel file is simply a Mach-O object file, much like any other object file output from the host platform's link editor. Thus, the available user level libraries provide ample support for parsing and examining the contents of such files. Mach-O object files consist of a header followed by a sequence of load commands to be issued to the run time loader; amongst these include commands to load segments into memory (LC_SEGMENT) as well as information regarding the static symbol table (LC_SYMTAB). Upon initialization, it is only the latter command that is of interest (individual segments and sections are examined later upon demand) to the MachLoader object. The typical Mach-O symbol table consists of several (in the case of the kernel file, hundreds) of symbol entries, each of which bounds a symbol (defined by its offset into a string table) to a virtual address. Again, care is taken to consider only symbols that are flagged as absolute (N_ABS) or as defined in a particular section within the object file (N_SECT). One must be careful to note that the section number (if any) reported by the symbol table is an ordinal flagging of the section relative to the start of the object file, and not relative to the start of any particular segment, as one might expect. As a final step in parsing the symbol tables, all symbol entries associated with a particular

section are stored into a vector, which in turn is inserted into a hash table keyed on the section ordinal. We then generate two additional hash tables that will be populated on demand. One of these tables facilitates the one-one lookup between a code address and the associated code fragment information, whereas the other provides the potentially many-one mapping between symbols and virtual code addresses.

Actual code lookup is initiated by an external source issuing commands to the MachLoader instance. Such commands typically request that a particular section, located within a certain segment, be parsed and analyzed. Examining the LC_SEGMENT commands mentioned earlier, the MachLoader object examines the referenced segments to determine whether or not the desired section is contained therein. Simultaneously, a section ordinal is maintained to track the section number relative to the start of the file; this facilitates lookup of the symbol vector bound to the desired section ordinal. Armed with this information, the loader instance may proceed to parse the code found within the target section. The MachLoader identifies code segments of interest by considering the range of addresses between two consecutive symbol table entries, or between the symbol table entry and the address at the end of the section (assuming, of course, that the symbol entries have been sorted according to their associated virtual address). Unfortunately, a few difficulties arise with direct application of this technique. First, symbols located in the symbol table include not only those of interest but extraneous symbols internal to the code segments as well (such symbols often correspond to the target of "goto" statements intended for error handling). Additionally, a sequence of symbol table addresses may share the same virtual address, thereby corresponding to a set of aliases for the same code segment. The latter problem is trivially solved by tracking all aliases while searching for the next contiguous symbol entry. Once found, the associated code segment is inserted under the symbol name given by the first alias (arbitrarily chosen) into the one-one hash table; however, a corresponding reverse mapping from name to code address is entered into the many-one table for each alias. In order to address the former problem, a distinction must be made between the primary code symbols and the useless internal symbols. Fortunately, the compiler provides such a distinction, though the natures of the naming conventions tend to differ by section. The two (segment, section) pairs of interest to the loader object are (__TEXT, __text) and (__VECTORS, __interrupts), the first of which is common to most UNIX-style operating system object file formats, whereas the second is peculiar to the PowerPC hardware. In the first case, primary symbols appear to be prefixed with a underscore ("_") whereas in the latter, primary symbols may be prefixed with either an underscore or a proper identifier (".L_"). Thus, prior to performing the code segment delineation operation outline above, the MachLoader object sorts the symbol entries for the target section first by label type (e.g., primary or not), and uses a sort based on the virtual address to order symbols belonging to the same class.

Once all desired sections have been analyzed using this technique, the code must be decompiled and the corresponding control flow and call graphs constructed. Our stipulation that a call graph be constructed (to facilitate analysis of system call chains in the future) forces an analysis of all code segments in the hash table, rather than simply the segments of interest (this is necessary as one would like to establish the identity not only of routines invoked by a code segment of interest, but also the identities of those code segments that invoke it). Waiting to perform this operation until all sections of interest have been loaded ensures that cross-section invocations are caught and recorded in the constructed call graph. Once this operation is completed, the MachLoader object may respond to code segment lookup requests by name using

a two-step lookup process (mapping name to address, and then address to code segment information).

Note that although the MachLoader object analyzes both the "__text" and the "__interrupts" sections, only the former is considered during dynamic code modification. As expected, this section contains the vast majority of kernel routines – all the routines contained therein abide by standard C-calling conventions (they accept parameters passed in registers, they access the stack relative to the stack pointer register, and so on). Code in the "__interrupts" section does not abide by such rules as they are executed at interrupt time by the PowerPC hardware. Unlike many other processors, the PowerPC hardware does not use a vector table to indicate the memory locations at which interrupt handlers are located; rather, the hardware assumes the primary handlers will always be present in the first two pages of physical memory at predetermined offsets (the booter guarantees that this code will be placed at the desired addresses). There are 256 instructions allocated to each exception vector, each of which is executed in privileged mode, with external interrupts and instruction and data translation disabled. Since the hardware spontaneously jumps to these memory locations in response to an exceptional condition, one cannot expect traditional ncalling conventions to be observed. Indeed, all registers (save some special supervisor registers) must be considered callee saved, requiring the system to take great care in saving (and then later restoring) the processor state at exception time. For most of the routines in the "__interrupts" section, the invariants (regarding e.g., available registers and enabled hardware features) tend to differ, complicating their analysis by the code analyzer, as well as reducing the likelihood that the client will introduce safe code conformant with the current variants into the operating system.

Another limitation of the current system is its dependence on the static symbol table found in the kernel object file. In microkernel-based operating systems particularly, additional services are dynamically loaded into the operating system using kernel extensions after boot time. The base kernel code is locked into low memory by the booter and is mapped one-one between virtual and physical, whereas extensions loaded dynamically are loaded into segments of virtual memory that may be pages. When performing the dynamic linking of the extension code into the kernel, the kernel loader adds the new module's symbols to the runtime kernel symbol table. According to Apple system engineers this runtime table is unavailable to user processes and is thus inaccessible to our current prototype. Future prototypes may augment the kernel loader to acquire access to this information.

3.2.2. Decompiler

Decompilation of PowerPC (PPC) machine code occurs in response to a request issued by the MachLoader object. Implementation of this module of the system, though lengthy, is fairly tedious, as it must carefully decode each opcode (and any accompanying extended opcode) in the PPC ISA. In proceeding to generate a suitable intermediate form representation for a PPC machine code instruction, the decompiler first classifies the instruction as belonging to one of the following broad categories: arithmetic, memory load, memory store, conditional branch, jump, or return. Each instruction class has a corresponding intermediate form representation that provides relevant information regarding the instruction. Furthermore, an inheritance hierarchy exists between the various instruction classes to further facilitate subsequent analysis. All instruction objects derive from a generic, un-instantiable root instruction class, which provides behavior for responding to generic queries (queries concerning, e.g. register usage or instruction location).

Next, the arithmetic, memory load, memory store, and conditional branch instructions form a second level in the inheritance chain. Unconditional branches (jumps) derive from conditional branches, and return instructions in turn derive from jump instructions. Constructing the class hierarchy in this manner enables the code analyzer to interrogate the instruction objects for more detailed information, once the class of the instruction is known.

A few difficulties arose in the implementation of this module as a direct consequence of the underlying hardware. Hardware substitution for register values constituted one such problem. In the event a particular instruction referenced general-purpose register r0 as a source register, the PPC hardware does not read the contents of register r0 but instead substitutes the hard-wired value 0 in its place. Though our system need not recognize this case for correct operation, it is beneficial to disregard such accesses as a use of register r0 in the hopes then that r0 will become available for dynamic code generation. In other cases, despite great care being taken to record the nature of all operands referenced by an instruction (be it a general-purpose register, floating point register, special-purpose register, constant, or the like), it was at times statically impossible to ascertain this information correctly. An example instruction is “stswx,” which spills a set of registers to memory, beginning with a specified source register all the way through a dynamically determined final register (based upon a value specified in the low order seven bits of the XER special-purpose register). The decompiler avoids incorrect analysis in such cases by conservatively assuming the worst case (in the example case, this would entail assuming that all general purpose registers are required by the instruction). Similarly, register indirect branches pose a similar difficulty for the decompiler. Often the code performs a register indirect jump to circumvent the branch displacement imposed by RISC ISAs; in such cases, improved code analysis can perform simple constant propagation analysis on the surrounding instructions to statically ascertain the target of the indirect branch. In other cases, the executing code loads the source register with a value read from a jump table or passed as a function pointer. Regardless of how much code analysis the decompiler performs, such values cannot be statically determined and such branch instructions are currently flagged as “indirect” by the decompiler (and subsequently ignored by the code analyzer). Finally, the decompiler uses a somewhat ad hoc method to determine whether a particular branch instruction is either a procedure invocation or a return instruction (the PowerPC ISA has no explicit return instruction, unlike the x86 or 68K ISAs -- save for the kernel’s return from interruption, or “rfi,” instruction). Observing the general expected convention that the link register is written by the hardware on function invocation, and that the invoked routine thus must issue a “blr” (branch to link register instruction) to return to its invoker, the decompiler classifies branches that write to the link register as “call” instructions and “blr” instructions as return instructions. Since these instructions Note that the technique described above will not recognize the pathological case in which a machine code instruction performs a PC-relative branch with link to the immediately following instruction (a trick used to determine the PC of the currently executing code), although this is perfectly acceptable for our purposes. A notable exception to the classification rule above is the user/kernel crossing instruction pair “sc” (invoke system call exception handler) and “rfi,” which despite their unconventional forms, conform to the expected semantics of procedure call and return, respectively -- at least insofar as the code analyzer is concerned.

3.2.3 Analyzer

Once the MachLoader instance generates a doubly linked list of instruction objects representing the machine code underlying a code segment of interest, it passes this instruction chain to the code analyzer for further analysis. The primary duty of the code analyzer is to conduct live register analysis on the kernel's machine code. Before the analyzer performs this data flow analysis on the intermediate form instructions, it must divide the instructions into basic blocks. A basic block is defined to be a sequence of instructions that -- barring system exceptions, context switches, or procedure calls -- execute in order. Instructions are partitioned into basic blocks using the scheme outlined in [1]. In particular, basic blocks begin with leaders, which include the first statement of the code segment of interest, the target of a conditional or unconditional branch, or an instruction that immediately follows a branch (either unconditional or conditional). A control flow graph may be constructed from the resulting basic blocks by linking them together according to the following simple rules: if a block B1 does not end in an unconditional jump, then the block B2 (which sequentially follows B2), or if a block B2's leader is the target of the branch at the end of block B1, then blocks B1 and B2 are linked together. In both cases, B1 is known as the predecessor and B2 the successor block.

Conducting a single linear pass over the instruction chain, the code analyzer will be unable to generate the complete set of basic blocks representing the code. In all likelihood, it may have already passed over the potential leaders of yet-to-be-created basic blocks. In order to address this problem, whenever a branch or jump instruction is encountered whose destination can be statically determined, the code analyzer creates a basic block division (as is to be expected), and also records the branching instruction along with a target address in a table. By performing a secondary pass over the entries in this branch target table, the code analyzer searches the master block list for a block containing the target address, and when found, subdivides the block to generate two new ones. In order to ensure correctness, some careful re-linking of basic blocks may be required. Using the technique described here, another search of the master block list must be performed in order to determine the basic block containing the source branch instruction. Note that simply using a cached pointer to the source basic block will yield incorrect results as the subdivision algorithm partitions existing basic blocks into two portions, reusing the original block's memory for the truncated upper portion. However, with a slight modification, the source block pointer may safely be cached in an effort to optimize basic block generation. It is guaranteed that the source instruction will always be the tail of some basic block (be it the tail of the original source block or some block derived from the source block); thus, the analyzer must simply check the original source block's tail instruction. If a match is found, the analyzer may conclude that the cached source block pointer is valid and may safely be used for basic block linking. If not, the analyzer can assume that it has subdivided the original source block, such that the tail instruction is located within some set of basic blocks descended from the original block. In this manner the number of blocks the analyzer must search to recover the desired source block may be sharply constrained.

Once the flowgraph has been built, the analyzer may proceed to conduct live register analysis upon the code. The goal behind conducting this analysis is to inform the end user which registers are safe for writing at the instrumentation site in the kernel. Though it is possible for the patcher to conservatively spill and restore all registers, irrespective of their liveness, the performance impact incurred for smaller patches may be unacceptable. In order to facilitate live register analysis, two additional basic blocks are introduced into the flowgraph: the entry node, whose sole successor is the code's start block, and the exit block, which has no successors but has its predecessors all blocks ending with a return instruction. Each block is retrofitted with a

bitvector representing the set of registers that are live immediately before and immediately after the code within a basic block executes. The code analyzer may then use this data to, on demand, determine which registers are free at a particular instruction address. This approach is preferable to expending valuable computing resources determining liveness information for each kernel instruction simply in anticipation of future use.

Live register analysis is a well known backwards data flow problem from compilers; one may find a thorough description of the technique in [1]. In essence, the data flow transfer function of a basic block is defined as the composition of the transfer functions associated with a single instruction, beginning with the transfer function of the block's tail instruction and ending with the leader instruction's transfer function. At the granularity of a single instruction, the data flow transfer function computes which registers contain live values before and after the instruction executes. For example, consider the PPC instruction: `or r3, r5, r3.` Further assume that only register r3 is live immediately the following the instruction. We then see that the set of registers live immediately preceding the instruction include r5 and r3. A formulaic way to derive this result, given the output liveness set, is as follows,

$$input[b] = (output[b] - def[b]) \cup use[b]$$

where "b" refers to the instruction under consider, "input" refers to the set of registers live prior to the instruction's execution, "output" refers to the set of registers live after the instruction's execution, "def" refers to the set of registers written by the instruction, and "use" refers to the set of registers read by the instruction. Using the composition of this data flow transfer function, similar information may be computed for each basic block. Since basic blocks and control flowgraphs introduce an element of non-serial execution, an iterative algorithm must be used to repeat the above calculations for certain basic blocks until the live register sets have stabilized. A possible future optimization to reduce the overhead of this iterative phase may involve imposing an ordering on basic blocks (such as reverse post-ordering) before computing their associated transfer functions.

The only difficulty that remains is initializing the live register analysis, i.e., which registers are initially considered live at the entry points of each basic block (including the entry and exit blocks). For the internal basic blocks, it is safe to assume that initially no registers are live; however, care must be taken when determining the set of registers live at the entry point of the routine's exit node, for this information will propagate backwards throughout the entire control flowgraph. In our case, the code analyzer initializes this set to be the registers the routine invoker expects to be intact following routine invocation (i.e., the callee-saved registers). Note this is certainly a conservative approach since it is highly doubtful that any routine would use all the allotted callee-save registers. Nevertheless, since the code analyzer cannot yet conduct interprocedural register usage analysis, this conservative approximation will suffice (though it may prove unnecessarily restrictive during our selection of free registers during code generation). An exception must be made however in the case of the return registers (r3 or f1/f2), which are caller-saved registers (a invoked routine is free to write the return value into these registers, so it is the invoker's duty to preserve their values if so desired). Despite this fact, the value contained in these registers may nonetheless be live upon routine exit. Reduced to working only with kernel machine code, the code analyzer does not have access to the routine prototype. As such it cannot know whether or not the routine is intended to return a value; it may err on the conservative side and assume all routines may return either a integer value (in r3) or a floating point value (in f1 or f1 and f2). Alternatively, a linear pass over the routine code will reveal whether or not the return registers in question are ever written to. For the purpose of our code

analysis any such write is sufficient to imply that the code segment may be trying to communicate a return value to the outside world. One final change made to the algorithm involves the inclusion of additional registers in the boundary set. Whereas the registers noted thus far are considered callee saved registers in user space, kernel code must be careful to avoid tampering with various privileged registers (including for example the memory segment registers). For this reason, a subset of the privileged registers one might be tempted to use is added to the set of registers deemed live upon routine exit. Again, as the crux of the analyzer's technique relies upon adherence to standard calling conventions, we see that our code analysis is not immediately applicable to the code found in the "__interrupts" section.

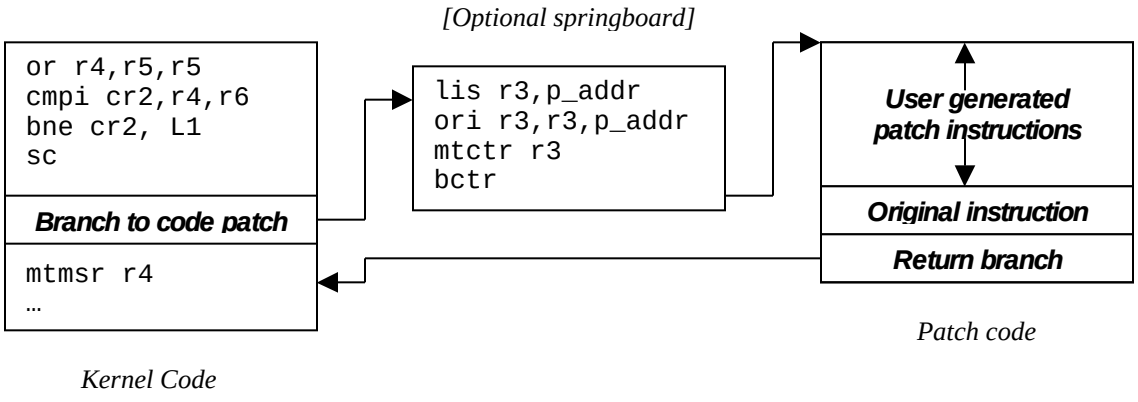
3.2.4 KernelInterface

The KernelInterface module is responsible for all necessary communication with the kernel component of the dynamic update framework. A variety of techniques may be used to communicate commands to the kernel component, including writing commands to shared memory buffers, invoking a system control call, or writing to a pseudo-file descriptor. Our current design uses the last method to accomplish this goal, largely due to the ease with which the corresponding kernel component could be implemented. In this scheme, individual commands are issued to the kernel component in the form of ioctls(), with the appropriate kernel response being communicated to user space via the errno variable. Currently, two commands are supported by the kernel component: patch kernel address and remove kernel patch.

In issuing the first command, the KernelInterface module first verifies several pieces of information regarding the instruction at the patch site. First, the KernelInterface module verifies that the code segment being patches adheres to standard calling conventions (i.e., that it is not interrupt code). Next, it determines the class of instruction at the instrumentation address, and disallows the patch procedure to procedure if the instruction in question uses PC-relative semantics. The reason for this is apparent once one considers the format of a dynamic code patch, depicted in Figure 2 below.

Figure 2: Dynamic Code Patch Format

.



Though the details of the patching mechanism are discussed in the Kernel component section below, for now it suffices merely to note that the original instruction is relocated to the code patch. As such, instructions that rely on PC-relative semantics will no longer yield the desired

results (the PC of the original instruction has been changed); in order to preserve correct operation, the kernel component would need to translate the original instruction into an equivalent sequence. Our current prototype does not currently implement this feature.

Once the interface object deems the original instruction safe for relocation to the code patch, it must compute the set of registers that are free immediately preceding the instrumentation point. These registers may then be used without explicit save and restore by the code patch. Computation proceeds using the live sets computed by the code analyzer; once the basic block containing the instrumentation instruction is found, a simple backwards flow analysis is performed on the basic block to ascertain the desired information (a simple inversion operation converts the set of live registers to the set of free, i.e., dead, registers). From Figure 2 above, the kernel component may need to allocate an optional springboard (described in the Kernel component section), which requires the availability of any general-purpose register and any register compatible with an indirect branch (either the link register or the counter register). If such registers are not available, then the patch is disallowed, erring on the conservative side (the kernel component determines whether a springboard is required only after the memory for the patch has been allocated).

Insofar as the actual patch is concerned, most clients would prefer to avoid generating the entire patch in assembler. A preferable alternative would be to insert a call to a routine written in a high level language, which has perhaps been loaded and linked into the kernel as part of kernel extension. Before invoking such a routine, the patch code must take care to save all caller saved registers to the stack (as the invoked routine is free to use these), and restore them upon return. By taking the difference between the system specified caller saved registers and the registers that are currently free at the instrumentation point, the KernelInterface instance determines the registers the patch code must save and restore. Since the compiler statically calculates the size of the stack frame used by executing kernel code, it is unlikely one will be able to locate space in the stack frame for additional registers spills. A solution to this problem involves the creation of a dummy stack frame below the current stack frame, solely for the use of the invoked routine. Once the invoked high-level routine returns, the patch restores any saved registers and removes the temporary stack frame, without perturbing the correct operation of the kernel code executing at the instrumentation point. Unfortunately, the host platform's runtime system present one additional complication -- the notion of a "red zone" beneath the currently active stack frame. As an optimization measure, compilers for the host platform eliminate the instructions needed to setup and tear down a stack frame from the code for leaf procedures (procedures which do not invoke other routines). Instead, leaf procedures may use up to a certain number of bytes (in the case of the Darwin runtime system, 224) below the stack pointer as temporary storage. To ensure correct operation of the kernel following patch installation, code patches that require stack access must be cognizant of such a "red zone" beneath the current stack pointer [3]. Using the call graph built by the code analyzer, it is possible to conservatively ascertain whether or not a given procedure is terminal in a call chain, and caution the generator of the code patch accordingly.

The final responsibility of the KernelInterface instance is to supply the kernel component with the user generated portion of the instrumentation code. This task may be accomplished in any number of ways, though for the sake of simplicity our current system works with user-entered assembler instructions. All of the user's input is spooled to a temporary assembly file, bracketed with directives to place the assembled machine code into the object file's "__text" section, with a specified alignment and symbol name. Executing a simple fork()/execve() pair, the KernelInterface instance proceeds to run the system assembler (/usr/bin/as) on the temporary

file, thereby producing an object file containing the desired machine code. Re-using our object file loader code from before (the MachLoader class), we are then able to successfully retrieve the machine code of interest. While convenient and simple in implementation, this approach suffers from key disadvantages, most notable being its reliance on the user to adhere to the host platform's runtime conventions (i.e., stack usage, register conventions, etc.). Further, impelling the user to construct the patch using low-level assembler code only increases the opportunity accidental introduction of error. An alternative solution would be to allow the user to code the patch in a high level language, and use a secondary code generation tool to output machine code using the set of available registers. A drawback of both techniques is the conspicuous absence of a kernel linking facility to resolve external references in the patch code; as a result, patch code entered by the user must explicitly refer to the addresses of kernel code and data (this is one benefit of using the patch as a call stub to a dynamically loaded and linked kernel routine). We hope to address this issue in a future revision of the system.

3.3 Kernel component

Whereas the bulk of the work takes place in user space, the final critical operations are performed by a relatively dense piece of kernel-resident code in response to commands issued from user-space. In our design it is the responsibility of the kernel component to ensure that the primary goals of the project (unrestricted dynamic update of kernel code and immunity to timing constraints) are met. We discuss some of the considerations, tradeoffs, and difficulties encountered during the process of developing the kernel component.

3.3.1. User space communication

It was imperative that we develop a simple but efficient means by which the kernel component could communicate with its user space counterpart. After some consideration, it was decided that the kernel component would publish a "pseudo-device" in the system /dev/ hierarchy with global read privilege. As such, any user space process can interact with this device node using built in file system commands, particularly the standard BSD open(), close(), and ioctl() calls. In order to avoid the issue of multiple user space components competing for access to this device, the kernel component grants exclusive access to user space processes on a first-come, first-served basis. The kernel component registers the device upon loading into the kernel and removes it from the /dev/ tree upon termination. Note that this technique, while perhaps not directly applicable in other operating systems, is likely to have equally simplistic parallels. Alternative to the pseudo-device scheme, as mentioned previously, include custom system calls, shared memory communication, or the like.

A KDRVPATCH ioctl() notifies the kernel that the client process has prepared a patch for dynamic insertion into the kernel. An accompanying parameter structure contains the target instruction address, a scratch general-purpose register, a scratch indirect branch register (either the link or counter register) for optional springboard generation, a pointer to a buffer containing the patch instructions, and a numeric count of the number of instructions in the buffer. In response, the kernel component allocates a segment of kernel memory large enough to accommodate the user instruction sequence, the original instruction (extracted from the instrumentation address), and a return to the instruction immediately following the instruction point. Note that, due to virtual memory protection, it may not be possible to directly read the

original source instruction from kernel memory. To circumvent this difficulty, the kernel component obtains the underlying physical address of the instruction (as the base kernel code is wired into physical memory), and conducts a raw physical read of the word at this byte. For our prototype, which deals only with base kernel code, this approach suffices; in the future, however, the associated memory page could be temporarily wired (and then unwired) in order to perform a similar read operation. Care must also be taken when accessing the buffer of patch instructions referenced by the parameter structure, as this virtual address refers to a memory location in the client's address space. Thus, specialized cross-address space routines must be used to transfer the data across the user-kernel boundary (such as the kernel uiomove() routine). Alternative strategies to transfer the instructions to kernel space could use a shared kernel-user space patch heap, managed entirely at the user level. Though such a technique would eliminate the overhead in cross-address space data transfer, management of the patch heap would be added to the already immense responsibilities of the user space code. Given the general size of the typical code patch, one may opt simply to leave this responsibility to the kernel's memory allocator. Having thus transferred the patch instructions into the kernel allocated space, the kernel component then writes the original instruction, followed by a return sequence, into the patch. Depending on the address of the patch in kernel memory and branch displacement limitations, the return address may not be reachable with a single branch instruction (in our case, however, the original instruction will be in the base kernel code and hence directly accessible via an absolute branch). Taking this possibility into account, space is allocated at the tail end of the patch in order to accommodate a multiple (in case of the PPC ISA, four) instruction sequence required to conduct a register indirect branch. Note the same registers allocated to springboard generation may be safely recycled here. Upon completion of patch generation, the patch is inserted into the kernel, and the relevant information (such as the instrumentation point, and pointers to the optional springboard and patch memory) is recorded in a hashtable in order to facilitate later removal.

The only other command accepted by the kernel component, KDRVREMOV, takes a single parameter representing a kernel instruction address. Using this passed address, the kernel component performs a lookup in its internal hashtable to ascertain whether or not the specified address had been previously patched. If a corresponding entry is found, the original instruction is restored at the instrumentation site, followed by deallocation of the memory associated with the patch and the optional springboard. Having thus restored the state of the kernel to its original form, the kernel component then purges the patch information from its internal hashtable before returning control to the client application.

3.3.2. Update Timing

An important constraint in any dynamic update system is the timing with which new code may be introduced into the target environment. Patching of code that remains inactive throughout the duration of the patch process is a safe operation; it is not possible to execute an unsafe mix of old and new code. The situation becomes considerably more complex when the instrumentation site is currently active, or soon to be active (by active, one typically implies that the instruction stream is being executed by a CPU thread). Overwriting multiple instructions at the instruction site may lead to race conditions in multi-threaded contexts, as threads may inadvertently execute a mix of old and new code, dependent on the OS scheduler's behavior and the quanta allocated to the various threads. Solutions that pause all executing threads (save the patching thread) and

perform stack introspection to determine whether any threads may be executing at or near the instrumentation point maybe applicable in the user context, but fail at the system level. In particular, pausing kernel operation generally interferes with I/O and exception processing, leading to erratic system performance and possible failure. When dealing with multiple instruction patching, one runs the additional risk of spilling code over the tail end of a basic block and into a successor block. If this occurs, an alternate entry path into the successor block (which does not pass through the patched predecessor block) will only execute a fraction of the patched code (this may also occur if a statically un-analyzable branch happens to jump directly into the middle of a patched sequence). Using single instruction splicing, all of the aforementiond problems may be safely avoided. By replacing the instruction at the patch site with a branch to patch code, we ensure that either the patch is executed in its entirety or not executed at all by any executing CPU thread. This allows for safe application of the dynamic patch technique in the fully multithreaded kernels found in most modern operating systems.

Unfortunately, the single instruction splice technique is only applicable if the inserted branch instruction occupies fewer bytes than the underlying instruction at the instrumentation point. In the case of RISC architectures, such as the PPC ISA, this is always the case as each instruction is guaranteed to occupy a fixed instruction width. With CISC architectures, the problem is more pronounced: as instructions occupy varying numbers of bytes, it is possible that an inserted branch instruction may cover multiple underlying instructions (the case in which the spliced instruction occupies less space than the underlying instruction is perfectly acceptable, as the tail end of the original instruction will never be seen by the instruction fetch hardware). A solution that is applicable in both the RISC and CISC contexts is the introduction of a single illegal instruction at the patch site, forcing the hardware to take an exception (in the case of the PPC hardware, this forces the CPU to jump to the fixed memory location 0x700 and execute the corresponding exception vector code). The corresponding handler may be augmented to note whether the exception stems from a dynamically patched address, and if so, the saved CPU state may be configured to point to the patch code when the interrupt handler exits. Note this approach stipulates that the host operating system be able to handle illegal instruction exceptions at any point during its execution, and further, that the code required to determine whether the faulting address is an instrumentation point must be safe to execute when the machine is in exception handling mode (e.g., interrupts disabled and virtual memory disabled), and must be safe to dynamically introduce into the kernel (if we choose to require the host operating system be an unmodified kernel). Even in the event these criteria are met, further research is required in order to determine whether or not the exception processing overhead associated with each dynamic patch proves to be prohibitive.

3.3.3 Cache Consistency

An idiosyncrasy of the PPC hardware, which may be present in other architectures, involves the issue of caching and dynamically generated code. In particular, the difficulty arises because the PPC maintains a split data and instruction cache, which lack any form of internal coherency. Dynamically generated code is written using standard load and store instructions, and as a result, remains resident in the hardware’s data cache (it is only lazily written to the underlying physical memory, as the cache uses a write-back policy). When the hardware subsequently issues an instruction fetch, a stale value may be fetched from the either the instruction cache or the underlying physical memory. Thus the responsibility of maintaining cache consistency between

the data and instruction cache falls upon software. One particular instruction sequence which addresses this issue is the following: `dcbf 0, r3` (assuming r3 holds an address containing dynamically generated code, this performs a flush to memory of the corresponding data cache line), followed by a `sync` (this enforces an ordering operation on issued instructions), followed by an `icbi 0, r3` (invalidates the instruction cache line referenced by r3), and finally an `isync` (to ensure all future operations execute in the context established by the preceding instructions) [10]. Since these instructions execute using the current virtual memory translation features of the processor, we need not worry that the caches themselves are physically addressed pieces of hardware. Care must be taken to ensure that all cache lines spanned by the dynamically generated code are handled by the above sequence. On the PPC, each cache line consists of 32 bytes (or eight instruction), so it is possible that valuable instructions will be flushed from the instruction cache by the `icbi`, forcing a redundant re-fetch of valid instruction data from memory. We expect the performance impact of this code to be negligible, however, as it is executed only once per dynamic patch introduced into the system.

3.3.4 Springboard Allocation

In some cases the spliced branch instruction is incapable of reaching the code patch due to branch displacement limitations. We conduct a simple test to determine whether an absolute or PC relative unconditional jump from the kernel code instrumentation point can reach the dynamic code patch, given the constraints of the branch displacement (28 bit signed displacement for the PPC `b` instruction). If so, then the appropriate branch instruction is spliced into the kernel machine code using the techniques discussed above, and the procedure continues as one would expect. The difficulty arises when the patch memory cannot be reached by the spliced branch instruction, and a more complex, register-based indirect branch must be performed. Note this scenario differs from the case in which additional instructions are required to return from the code patch; there, the kernel component may freely allocate additional space to accommodate the longer instruction sequence. Since space exists to splice only a single instruction into the kernel code (in order to avoid update timing restrictions, as discussed above), the indirect branch sequence must be located in a segment of memory accessible using the standard branch displacement. Although the program exception based technique mentioned earlier manages to avoid this issue (as the kernel may return from the exception directly to the code patch, regardless of its displacement from the base code), the simplicity afforded by the single instruction branch method warranted further consideration of the branch displacement problem.

The obvious solution was to allocate “springboards”, or small segments of memory, from low virtual memory addresses, as these are guaranteed to be reachable using standard PPC absolute branches. Unfortunately, no clean and portable general solution could be found; the techniques used to acquire low memory are completely ad-hoc and dependent on the architecture of the underlying host hardware and operating system. Our preferred technique of requesting the kernel to allocate a large region of virtual memory (two to three pages) in a suitable address range could not be guaranteed to be succeed with high reliability, as was determined through a communication with Apple’s operating system engineers. Thus, in the process of developing a suitable technique for obtaining the required springboards, our attention turned to the unused addresses in the first two pages of memory, where the primary exception vectors resided. Though the hardware allocated most, if not all, of these exception vectors 256 bytes to process an

exception, the vast majority of these used no more 20 bytes of storage. By grouping together the spare memory areas littered throughout the exception vector code into a set of arenas, a dynamic memory allocation library was written to selectively allocate and deallocate blocks from these arenas. To circumvent virtual memory protections imposed by the operating system on these code pages, a spare region of kernel memory with full read write privilege was allocated and manually mapped to the underlying physical addresses of the vector code pages. These pages in turn comprised the memory pool backing the customized memory allocation library, which in turn was capable of dynamically allocating up to 173 springboards. However, when converting a aliased memory addresses into a corresponding low memory address (for use in generating an absolute branch instruction aimed at the springboard), the tacit assumption that all base kernel virtual addresses were mapped one-one with wired underlying physical memory proved to be incorrect.

In an effort to catch null pointer writes within the kernel, the bottom two pages of kernel memory are stripped of all read/write privilege. Unfortunately, as dictated by the PPC memory management scheme, page protections are determined as a combination of protection bits in the appropriate page table entry and the associated segment descriptor (the upper four bits of any virtual address on the PPC are used to select one of sixteen segment descriptors, which contain additional address space information). For the kernel's address space, the protection key is defined to be the privileged bit in the segment descriptor, which happens to have the value of zero. Given this protection key, the absolute minimum possible access rights to a page is "read-only." "No access" may be specified only if the segment protection key is changed to a high value, though enacting such a change will modify the permissions on 2^{16} pages of kernel virtual memory. Without introducing such a change, one must enforce restricted access by leaving the pages unmapped in the kernel's physical memory map (or mapped to physical address zero, which contains invalid instructions). This effectively renders the solution proposed above useless, as the memory management hardware must be capable of translating the address of the springboard and executing the code located therein. To overcome this impasse, a compromise was made: of the two low memory pages left unmapped, the second's virtual address range was reclaimed (0x1000-0x2000) for use by the springboard allocator. Though the magnitude of the available scratch region appears to be reduced, the allocator has free reign to use the entire address range as it pleases -- the corresponding physical addresses are in no way bound to the virtual addresses of interest. As such, the allocator, upon initialization, allocates a page of wired virtual memory, and redirects the unmapped virtual memory region to refer to the underlying physical memory underlying the newly allocated page. Using this technique, the allocator is able to successfully allocate up to 170 springboards for use in dynamic code insertion, all of which are accessible from virtually any instruction in the entire kernel address space, using a single absolute branch.

4. Results and Applications

4.1 Results

The implementation complexity of the dynamic update system exceeded our initial estimates; indeed, the author of the dynamic kernel instrumentation system for Solaris [17, 18, 19] had forewarned us that an implementation built from scratch would likely consume the better part of a year. Due to the fact that the KernInst project source was unavailable to the public, an

implementation of a prototype for our own research purposes was inevitable. Fortunately, we were able to develop a running prototype of our system in a greatly reduced period of time, partly due to the months of relevant industrial experience we brought to the project. However, though we have succeeded in retrofitting a modular dynamic update system on top of a commodity microkernel-based operating system, we are still in the process of exploring some of the research questions we initially posed. Below is a brief overview of some of the cases we have begun to examine.

4.2 Example application: Dynamic Extensibility

In the Darwin operating system, network packets are received from the network driver and demultiplexed by packet type (e.g., NETISR_IP, NETISR_ARP, and NETISR_IPV) to the appropriate packet queue. A bit is flagged in a 32-bit integer (netisr) to indicate which packet queue needs servicing by the primary network thread. When scheduled, the network thread runs a software interrupt routine (run_netisr) to process the bits set in this integer, and service the referenced queues in some predetermined order. There are a few notable shortcomings to this scheme: the number of packet types is constrained to the host integer width (32), and the packet types checked for and processed are statically determined. Without necessitating a kernel recompile and reboot, it is possible to explore the performance consequences of moving to a callback based system, in which run_netisr() invokes for each flagged packet type a routine previously registered with the kernel by the user. Another compelling example involves the addition of additional packet types dynamically (via a kernel loadable module). Perhaps for performance reasons, the loaded module would like its packet type and associated packets serviced during the CPU quanta allocated to the network daemon. Furthermore, by observing the in-core machine code of the kernel, the module may dynamically select an index number for its packet type, so as not to conflict with any other packet servicing routines. In many cases, one would prefer this line of action to the notion of the dynamically loaded module creating a high priority kernel thread to service its own packets. Indeed, such a thread would compete with the primary network thread for CPU resources, only adding additional context switch overhead to the packet processing process. Although with modern CPUs the associated context switch overhead may appear minimal, in high-end, throughput-intensive applications, it may prove to be a bottleneck. Note that dynamic modification of the kernel's run_netisr routine addresses both of these concerns.

A trivial example, which demonstrates the applicability of our system, is a runtime modification of the run_netisr routine to invoke a sample logging routine if any bit in the netisr is set. Assume that our sample logging routine (shown below in Figure 3) is dynamically loaded and linked into the kernel at address 0x15628f5c. Then the machine code for run_netisr may be analyzed, and a dynamic patch of the form noted in Figure 3 below may be inserted at address 0x000dbdb8. Upon successful completion of this procedure, observing the system log (/var/log/system.log) in the wake of any network activity should yield the output shown below (also in Figure 3).

Figure 3: Dyamic Kernel Extensibility

Initial source code:

```
extern int netisr;
```

```
static void packet_counter(void)
{
    static int counter = 0;

    counter++;
    printf("KernMonitor/packet_counter: netisr value: 0x%08x | %d\n", netisr,
counter);
}
```

Patch code:

```
lis r3, 0x1562          ; This short assembly sequence serves
ori r3, r3, 0x8f5c      ; as glue to invoke the above routine.
mtctr r3
bctrl
```

System log output (after patch):

```
May 08 05:36:05 makaveli mach_kernel: KernMonitor/packet_counter: netisr
value: 0x00000004 | 19
May 08 05:36:05 makaveli mach_kernel: KernMonitor/packet_counter: netisr
value: 0x00000004 | 20
…
```

4.3 Example application: Layer Optimization in the Mach Memory System

The Mach microkernel has taken great care to separate its memory management portion into machine independent and machine dependent portions, with the hope of facilitating system portability. In particular, the machine dependent module (the pmap layer) implements the operations necessary to manage (in our case) the PPC memory hardware unit and its associated page tables; all remaining virtual memory information is managed by the machine independent portion [16]. When considering the set of exported pmap routines in the Mach microkernel, it is interesting to speculate whether the machine independent and dependent layers may be dynamically collapsed to produce a positive affect on system performance. With the aid of our user space tool, we have discovered frequent call chains such as vm_fault -> pmap_enter -> mapping_make -> hw_add_map, which tend to cross the machine dependent/independent boundary whilst traversing the various layers of the memory system. We are currently adding dynamic instrumentation to these call chains in order to gain insight into any benefits dynamic optimization might provide.

5. Conclusions and Future Work

Our research efforts have succeeded in developing a functional prototype of a modularized, fully dynamic kernel update scheme on top of a commodity microkernel-based operating system. While we have examined some trivial applications of this technology, we are still exploring the avenues for the possible incorporation of dynamic optimization techniques such as aggressive code inlining and constant propagation. As the Synthesis research and our preliminary results have tended to suggest, microkernel-based operating appear to provide fertile ground for successful application of such technologies.

Despite the implementation-related limitations discussed throughout the paper, many issues remain. One such issue of interest is the portability of the system to an x86 (CISC) based architecture; based on the arguments given in this paper, an implementation on such hardware should be technically possible, albeit slightly more complex than the implementation on its RISC counterparts. Another issue worth some consideration is the issue of security and safety with regard to dynamic code insertion into a running kernel. What are the trust policies which accompany such technology, i.e., who should be allowed to conduct such changes? How can the correctness of dynamically generated code be verified by the host operating system? One possible solution to this latter problem involves the use of run-time code augmentation (similar to the run-time checks conducted by the Java runtime system). Additionally, the effects of errant code may be minimized using the software fault isolation techniques employed by the VINO or Exokernel projects, though this becomes a difficult challenge when the effects of the dynamic code cannot be easily localized (i.e., the code accesses and modifies global kernel state). The question of whether or not proof carrying code (PCC) may be feasibly used to ensure type safety in the context of dynamic operating system modification remains open as well. Such unresolved topics provide ample ground for future expansion to our initial work.

Acknowledgements

I would like to thank Professor Monica Lam for providing direction and support over the course of conducting this research. I would also like to extend my thanks to Vinesh Patel, Ranganath Sudarshan, Sumir Meghani, and Raymond Chan for their thoughts and commentary on the material presented in this paper.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1988.

[2] Apple Computer. *Loading Kernel Extensions at Boot Time*. **http://developer.apple.com**, 2001.

[3] Apple Computer. *Mach-O Runtime Architecture: Loading and Linking Code at Runtime*. **http://developer.apple.com**, 2001.

[4] Apple Computer. *Network Kernel Extensions*. **http://developer.apple.com**, 2000.

[5] The Darwin Operating System open-source project. **http://developer.apple.com/darwin/**.

[6] M. Hicks, J. Moore, and S. Nettles. Dynamic Software Updating. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 2001.

[7] M. Hicks. Dynamic Software Updating. Thesis proposal, University of Pennsylvania, September 1999.

[8] How to Add a Character Device to Darwin. **http://www.darwinfo.org/howto/chardev.shtml**.

[9] K. Loepere. *Mach 3 Kernel Principles*. Open Software Foundation and Carnegie Mellon University, 1992.

[10] Motorola. *MPC750: RISC Microprocessor User's Manual*. Motorola, 1997.

[11] Motorola. *PowerPC Microprocessor Family: The Programming Environments for 32-bit Microprocessors*. Motorola, 1997.

[12] NetBSD Documentation: Writing a pseudo device. **http://www.netbsd.org/Documentation/kernel/pseudo/**.

[13] C. Pu,, H. Massalin, and J. Ioannidis. The Synthesis Kernel. In *Computing, Springer Verlag (Heidelberg, FRG and NewYork NY, USA)-Verlag Systems*, 1988.

[14] C. Pu and J. Walpole. *A study of dynamic optimization techniques: Lessons and directions in kernel design*. Technical Report OGI-CSE-93-007, Oregon Graduate Institute of Science and Technology, 1993.

[15] C. Pu, J. Walpole, and H. Massalin. *A Retrospective Study of the Synthesis Kernel*. Oregon Graduate Institute of Science and Technology, 1993.

[16] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31--39, Palo Alto CA, October 1987.

[17] A. Tamches and B.P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, February 1999.

[18] A. Tamches and B. Miller. Using Dynamic Kernel Instrumentation for Kernel and Application Tuning. In *The International Journal of High Performance Computing Applications*, 1999.

[19] A. Tamches and B. Miller. Dynamic Kernel Code Optimization. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. September 2001.