Process-Level Virtualization for Runtime Adaptation of Embedded Software

Kim Hazelwood
Department of Computer Science
University of Virginia

ABSTRACT
Modern processor architectures call for software that is highly tuned to an unpredictable operating environment. Process-level virtualization systems allow existing software to adapt to the operating environment, including resource contention and other dynamic events, by modifying the application instructions at runtime. While these systems are becoming widespread in the general-purpose computing communities, various challenges have prevented widespread adoption on resource-constrained devices, with memory and performance overheads being at the forefront. In this paper, we discuss the advantages and opportunities of runtime adaptation of embedded software. We also describe some of the existing dynamic binary modification tools that can be used to perform runtime adaptation, and discuss the challenges of balancing memory overheads and performance when developing these tools for embedded platforms.

Categories and Subject Descriptors
C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems—Real-time and embedded systems; D.3.4 [Programming Languages]: Processors—optimization, run-time environments

General Terms
Design, Experimentation, Measurement, Performance

Keywords
embedded systems, virtualization software, runtime adaptation, dynamic binary optimization

1. INTRODUCTION
The recent proliferation of processor architectures that incorporate multiple and potentially heterogeneous processing elements has resulted in an unintended consequence. More now than ever before, software developers must factor in the fine details of the underlying architecture in order to achieve improved performance on the latest generation of microprocessors. Furthermore, runtime inefficiencies inevitably arise due to the unpredictable environment that a given application is exposed to at runtime. For instance, the fact that multiple processing elements share resources, such as caches or buses, means that significant contention may arise at runtime that a software developer could never have predicted, and in fact, software development time is the wrong time to try to predict such runtime contention. In addition, the specific load on any accelerator or graphics processor will often determine whether a given calculation should occur on the main CPU or any accelerator hardware. What we want is software that is capable of adapting to the runtime environment that it encounters, and system and hardware support for making this runtime adaptation possible.

While the general-purpose computing community has begun to embrace the importance and potential of runtime adaptation, the need to adapt to the runtime environment is especially important in the case of embedded systems. Aside from the new complexities introduced by multiple cores and heterogeneity, several longstanding challenges have always existed on embedded systems that make it the ideal platform for adaptation. Power, energy, and battery life concerns are often at the forefront of the challenges for embedded systems, and these concerns often trump performance as a design constraint. These factors are in a constant state of flux. Meanwhile, at the software level, the specific implementation of a software application tends to be fixed at runtime regardless of the dramatic changes that may occur in terms of energy or battery life. Often, it is possible to implement an alternative, lower energy calculation in place of the standard implementation, but developers would not want to (and should not) implement the lower energy option permanently, as it would undoubtedly suffer performance consequences, which will always remain a concern, often for correctness reasons (e.g. real-time applications).

Process-level virtualization [10] is a promising opportunity for enabling runtime adaptation of software. By placing a virtualization layer between the running application and the underlying system (be it an operating system or simply bare metal), there is the opportunity to inspect and potentially modify every instruction that executes on the system. This includes shared library code, dynamically-generated code, and even self-modifying code. Changes to the code can be as simple as inserting extra profiling instructions or as complex as translating all instructions from one architecture to another. Meanwhile, all of these changes can be made trans-
parent to the application such that any introspection provides identical (or at least similar enough) results to a native execution, including bug-for-bug compatibility.

Virtualizing a single process at a time has several distinct advantages over virtualizing the entire system and all running applications (below an operating system, for instance). The dynamic compilation oriented approach means that the virtualization layer has a global view of the code of the running application, including information about potential execution phases and events, even before they occur. The ability to focus on application-specific profiles combined with environmental information simplifies the task of adaptation.

Runtime adaptation using process-level virtualization comes with a set of distinct challenges, however, and the challenges are especially acute in the embedded systems domain. The act of modifying any application requires processing resources, and when the modification is performed at runtime, this modification competes for resources with running the application itself. This overhead can be difficult to overcome. As a result, the motivation for adapting the software must be worth the overhead incurred to perform the adaptation, and only in rare cases will a motivation of performance improvement alone actually pay off. The memory footprint of the virtualization software is another major challenge. In many cases, the virtualization software has been known to greatly exceed the size of the guest application, particularly since many known optimizations trade memory overhead for speed.

Even for the case where the virtualization system can be designed to be compact and efficient enough for an embedded platform, an additional challenge exists that is specific to systems that virtualize at the process level. A system that supports parallel multitasking will potentially have multiple applications and therefore multiple virtualization engines running at once. This will result in significant memory and performance overheads that make it difficult to scale the solution to a large number of concurrent applications on multiple cores (although an argument can be made that scaling along this dimension is a low priority for embedded systems, at least for the near future).

Along with each of these challenges is a set of unique opportunities. It is much more feasible to provide hardware and system support for runtime adaptation in the embedded domain than it is in the general-purpose computing domain for a variety of reasons including, but not limited to, the rapid hardware design cycle. In the general-purpose community, there is significant pressure for virtualization systems to run on stock hardware and systems software. In fact, the ideal case of developing adaptive systems by co-designing the hardware and software layers would be far too disruptive for general-purpose computing, while it is the norm for embedded systems.

The remaining sections of this paper elaborate on the benefits and challenges of building adaptive software on embedded systems. Section 2 discusses some motivating cases where adapting software to the operating environment can be very beneficial. Section 3 discusses some software systems that make runtime adaptation of running processes possible. These systems perform dynamic binary modification to achieve the goal of adapting legacy software. Section 4 discusses several of the challenges encountered when building dynamic binary modification tools for embedded systems, while also presenting some of the solutions that have been discussed and/or implemented. Finally, Section 5 summarizes and makes the case for additional research into adaptive solutions for embedded systems.

2. ADAPTING EMBEDDED SOFTWARE

Building software that can adapt to an ever-changing runtime environment is particularly important for embedded systems. Battery life, power considerations, security, and code compatibility between dramatically changing architectures are all first order design constraints in this domain. In this section, we describe several scenarios that call for adaptive software solutions.

2.1 Shared Resource Contention

Most modern architectures incorporate several processing elements on the same die, and the resulting processors are termed multicore processors. One feature of nearly all known multicore designs is that they incorporate shared structures between the multiple processing cores. Some cores will share an L2 cache; others will share a bus. The problem that arises is that running applications can now be significantly affected by the other applications that happen to be running on a given machine. Meanwhile, the application was designed and optimized assuming an unloaded system (e.g. the memory and cache placement has not been designed to play nicely with other applications). While it would be possible to build software that plays nicely with other applications, we really want software to be greedy whenever it can, and to play nicely when it has to.

Figure 1 demonstrates the significant impact of resource contention in multicore systems. For the SPEC 2006 integer benchmark suite, the performance of a particular application can vary by up to 3X depending on whether the application was run in isolation or it was run alongside other applications. Note that all applications were “pinned” to their respective cores, so the applications were not competing for the cores themselves, but instead for the shared structures between the cores. (While these particular results were
gathered on a general-purpose machine, resource contention should be expected to occur on any processor where structures are shared between cores.)

Since it is nearly impossible to predict, at design time, whether (and to what extent) other tasks will be running alongside a given application, the issue of resource contention should be managed at runtime. Meanwhile, an ideal place to manage resource contention is in a layer that is acutely aware of the details of the running application.

2.2 Processor Heterogeneity

The challenge of adapting to multiple processing cores is difficult enough even if we assume identical capabilities between the various cores. The world gets even more complicated, however, if we consider that those processing cores may vary in several dimensions [5]. Heterogeneity is cropping up in modern multicore systems for two fundamental reasons. First, manufacturing defects often result in seemingly identical cores that in reality have wildly varying behavior in terms of performance, power, or even correctness. Figure 2 illustrates the temperature heterogeneity observed between cores on the same chip – core 0 consistently ran hotter than core 1, even though temperatures should have been identical. Second, many multicore systems are being designed with one or more specialized processing elements or accelerators, often right on the same chip.

For both causes of processor heterogeneity, the challenge is scheduling tasks on the available processing resources without a priori knowledge about availability and capability. It also becomes necessary to ensure that the software instructions themselves support relocation from one core to another, particularly in the case of accelerators that feature a different instruction set than the main core. While static assignments are possible, they are not resilient to the effects of runtime occupancy and contention for processing resources. Meanwhile, shipping multi-versioned code that is capable of running on either device is likely to suffer from memory consumption concerns. A runtime adaptation engine can dynamically regenerate the instructions to run correctly and efficiently on a variety of hardware designs.

2.3 Power-Aware Computing

Battery life, temperature, and reliability concerns are all first class design constraints for embedded systems, and all can be categorized as a form of power-aware computing. Unfortunately, it is difficult if not impossible to predict whether any of the concerns will become critical when designing and optimizing software. Meanwhile, there are steps that can be taken at the software level to mitigate each of these concerns. For instance, it is possible to generate instruction sequences that are more stable than others [22], which will improve reliability, possibly at the expense of performance. Given the potential performance degradation, it is clear that such code should only be generated when reliability concerns reach a certain level, which will generally not be known before runtime. Similar code generation trade-offs occur for addressing temperature or battery life concerns, but again, the need for such code is best deferred until runtime. By dynamically adapting the software on an as-needed basis, it is possible to achieve the best of both worlds.

2.4 Privacy and Security

Program shepherding is an effective technique for observing the runtime behavior of software, detecting anomalies, and enforcing a set of rules that prevent common exploits [16]. This technique works by inspecting each instruction prior to executing it, and determining whether it follows a suspicious pattern, such as writing to the stack. Since the technique can be applied to an unmodified program binary, it has been leveraged by several industrial products.

On an embedded system, security and privacy are equally if not more important than on a standard high-performance platform. Therefore, applying program shepherding in the embedded domain can be quite beneficial, assuming that it is still possible to provide full code coverage for the dynamic paths, and that the performance and memory overhead is not prohibitive. The former assumption can be proven for most architectures, while the latter assumption is still an open question that will be discussed in Section 4.

2.5 Code Compatibility

New processors are regularly released with extensions to the instruction-set architecture. This presents several challenges to both architects and software vendors. First, architects need an effective way to measure the utility of any new instructions they propose, prior to committing to the actual hardware changes. For instance, it is often helpful to design and optimize the compiler algorithms that would be used to generate the new instructions before committing to building the hardware to support those new instructions.

Unfortunately, having a compiler actually generate the new instructions means that binary cannot be executed until the hardware is available. Otherwise, any attempt to run the program will result in an illegal instruction fault. This catch 22 situation can be resolved by adapting the software to recognize and emulate the behavior of any new instructions that are unsupported by the underlying hardware. At the same time, the application can be adapted to introduce profiling code to determine the dynamic instruction count and other metrics of the new instructions (as discussed in Section 2.6), which will provide valuable feedback to the ISA architecture and compiler teams.

Once new instructions have been approved and introduced to the ISA, a second challenge is faced by software vendors,
who must now decide whether to include those instructions in their shipped applications. Including the instructions can significantly improve performance on systems that have hardware support for those instructions. Yet, illegal instruction faults will occur on systems that do not support the instructions. An elegant solution to both of these challenges is to adapt the software at runtime. This technique enables backward compatibility, allowing application binaries that contain new instructions to execute on older hardware that does not support those instructions. Meanwhile, it also provides an opportunity for forward compatibility by introducing new instructions to existing binaries on-the-fly, improving the performance of applications running on newer hardware that contains the ISA extensions.

Takens to the extreme, a similar runtime adaptation engine could provide full binary translation capabilities, allowing software to run on a completely incompatible platform.

### 2.6 Program Analysis and Debugging

Understanding the dynamic behavior and bottlenecks of running software is important for software developers and system designers alike. An effective way to achieve this goal is to modify the running program to report any feature of interest, such as instruction mix profiles, dynamic code coverage, or memory allocations and deallocations. Yet, building these features into the source code of the application is problematic for several reasons. Aside from the fact that the process would be tedious and error prone, it would also be deceptive because it would miss all of the time spent executing instructions in shared library routines or dynamically-generated code. Therefore, a great way to analyze a program’s dynamic behavior is to modify the program’s execution stream to interleave profile-collection and analysis routines that will be triggered as the program executes. Software profiling and debugging is critical for embedded systems software, and the same infrastructures that provide runtime adaptation can also be used as robust and thorough program analysis tools.

### 3. Runtime Adaptation Tools

A number of virtualization tools exist today that enable running software to be adapted by an external user. These tools exist both at the system level (a single copy that runs below an operating system) and at the process level (one copy per guest application that runs above an operating system, if present). Focusing on process-level virtualization tools, most run exclusively on general-purpose architectures [2, 4, 17, 18, 20, 21] such as x86. However, a few exceptions exist, and several tools have been ported to run on resource-constrained or embedded architectures. The Pin dynamic binary instrumentation tool, for instance, runs on ARM [11] and the Intel ATOM. Meanwhile, the Strata dynamic binary translator runs on ARM [19]. And finally, DELI runs on the LX architecture [7].

The process-level virtualization systems listed are often categorized as dynamic binary modifiers, virtual execution environments, or software dynamic translators, depending on the preference of the particular researcher, but they all operate in a similar fashion. Each system operates directly on the guest program binary, with no need to recompile, relink, or even access the source code of the guest application. This allows for even legacy or proprietary software to be adapted to the runtime environment.

![Figure 3: A runtime adaptation engine will continuously monitor and adapt the program image (often caching the modified code for efficiency).](image)

The virtualization system begins by taking control of execution before the guest program launches. Next, it inspects and potentially modifies every instruction (or series of instructions) in the guest application just prior to executing that instruction. The system is able to modify shared library code, dynamically generated code, and even self-modifying code — everything except kernel code, since it operates at the process level.

Process-level virtualization systems can naturally be leveraged as a runtime adaptation engine, since they allow software to be continuously profiled and modified, as shown in Figure 3. If a region of code is determined to be frequently-executed and in need of adaptation, the system will alter the code and will begin to execute the altered code in lieu of the original code. Finally, to improve performance, the system will often cache the altered code for reuse during a single dynamic run. The fact that these systems operate dynamically (at runtime) means that only the portions of the software that executes will be modified.

### 4. Implementation Challenges

Performing software adaptation at runtime requires the use of a toolkit that can be complex to develop. Achieving correct functionality in the toolkit is relatively straightforward — there are a variety of known techniques that have been leveraged by similar systems to ensure robust functionality. The true challenge lies in making the system run efficiently, particularly in a resource-constrained environment.

#### 4.1 Maintaining Control

One of the first goals a runtime adaptation engine must ensure is that it maintains complete control of the execution of a guest application, from the first to the last instruction. Security applications, for instance, require that every instruction is inspected and potentially modified prior to executing it. Challenges include acquiring control (injecting the adaptation engine into the guest application) and maintaining control across branches, calls, and runtime events. Indirect branches — where the branch target is stored in a register or memory and thus it varies at runtime — are particularly challenging to handle efficiently [8, 15] and on architectures like the ARM, any instruction can write to the program counter register, which ultimately results a branch. Finally, self-modifying code presents a challenge on most platforms, but is actually straightforward to handle on architectures like ARM that require explicit synchronization instructions whenever instructions are overwritten [11].
4.2 Balancing Memory and Performance

Another design decision faced by developers of a runtime adaptation engine is whether the engine will behave like an interpreter or like a just-in-time compiler. Interpretation-based approaches perform a lookup into a table that indicates the behavior and added functionality required for each instruction, and this approach works best for short-running programs. It has a higher performance overhead, but a lower memory overhead. Just-in-time compilation approaches will generate new versions of the code with all extra functionality inlined, and will execute that new code in lieu of the old code. This approach works well for longer-running applications where the cost of transforming the code will be amortized.

For JIT-based approaches, the next decision is whether to include a code cache in the design. The code cache will store previously modified code to facilitate reuse throughout execution, and the use of a code cache has been shown to improve performance by up to several orders of magnitude. However, the memory overhead of a code cache is significant as well—sometimes up to 5X the memory footprint of the guest application [12]. The high memory overheads are the result of the need to maintain a complete directory of the code cache contents and certain features of the resident code, the need to incorporate trampolines and other auxiliary code to maintain control, and the need to track whether the code has been patched in any way. The memory overhead pays for itself in performance improvements on general-purpose systems, but the balance on embedded systems is much more intricate, and therefore has required special attention [9].

Within the code cache, one design challenge is how to store the modified code. Most systems form traces—single-entry multiple exit regions—because they are more conducive to optimizations, and they tend to improve performance over storing individual basic blocks. Caching traces uses more memory than caching basic blocks, but it allows fewer entries to be present in the code cache directory (because multiple blocks can be encompassed into one trace) and the directory size reduction saves memory. With the formation of traces comes the challenge of trace selection [1, 6, 13, 14]. The longer the trace, the less likely the tail blocks will be executed, but when the speculation is correct, a longer trace will result in a performance boost. But of course, longer traces use more memory.

Finally, with any bounded size code cache comes the challenge of handling cache eviction and replacement policies [3, 9, 12, 24, 23, 25]. Standard hardware replacement strategies do not apply because the eviction unit is a trace, which varies significantly in size. Therefore, a traditional LRU policy cannot be implemented since the LRU element may not free enough space and thus a contiguous victim trace must be taken as well. While general-purpose CPU implementations have been able to support features such as unbounded code caches, this is not an option on an embedded platform.

Each of the challenges discussed in this section represent a standard memory-performance tradeoff. Most runtime adaptation engines have been developed for general-purpose systems, and therefore the memory-performance balance tends to be skewed toward performance. At the extreme, this means that they occupy too much memory to work at all on an embedded platform. However, with careful tuning and additional research, these systems can be made to execute efficiently in both domains. Therefore, it is important for embedded systems researchers and practitioners to be aware of the adaptation opportunities enabled by these systems, and to also be willing to invest their time, resources, and creativity to efficiently design and/or leverage these systems.

5. SUMMARY

Runtime adaptation is a promising opportunity to resolve many pressing computing challenges today, including resource contention, power, code compatibility, and security. The need for adaptive software is especially true in the case of embedded systems, where each of these concerns is magnified. Several process-level virtualization frameworks are available that enable effective runtime adaptation of software. Unfortunately, a set of key challenges remain that have prevented widespread acceptance of these frameworks by the embedded systems community. Most of the major challenges fall under the realm of the standard memory-performance tradeoff. In the general-purpose computing domain, developers have tended to favor performance at the expense of memory consumption. However, such an approach is inappropriate in the embedded domain, and there is still much work to be done to find that ideal balance.

Acknowledgments

This work is funded by NSF Grants CNS-0964627, CNS-0916908, CNS-0747273, and gift awards from Google and Microsoft. The implementation of Pin for ARM was a group effort involving several people including Artur Klauser, Robert Muth, and Robert Cohn. Several of my students contributed to this paper: Dan Upton gathered the resource contention and temperature heterogeneity data, and Apala Guha developed several of the memory-management algorithms for ARM. Finally, I wish to thank the organizers of this DAC special session for providing this opportunity: Christoph Kirsch and Sorin Lerner.

6. REFERENCES


