Statement of Research Interests
Arrvindh Shriraman

Computing systems have significantly evolved over the years, and have come to occupy a major part of our daily lives. Hardware designers have fueled this growth by doubling performance every two years. Meanwhile, software has continued to include more features and has increased in complexity. It has become challenging to develop correct, high performance, and reliable programs. Approximately 10 new bugs are reported every day for popular applications we have come to rely upon (e.g., Firefox). Multicore processors have exacerbated the challenges — algorithms, languages, and operating systems have all been forced to adopt parallelism, which increases the likelihood of concurrency bugs.

The overall goal of my research is to utilize some of the transistors afforded by Moore's law to develop new hardware mechanisms that lead to better software development tools and programming models. The basic idea is to use hardware to expose information that enables a program to understand its own execution and react to it. This will help developers understand misbehaving software. For example, if software could easily track the locations accessed by a program, it can use this to detect concurrency bugs and check safety invariants. Similarly, if software was aware of the cached locations, it could reorganize the code to overlap computation with cache misses and improve performance. Such information is difficult to get with the current set of tools or they impose high overhead. A distinguishing aspect of my work is that it develops hardware mechanisms that directly interact with software and builds infrastructure across the hardware-software interface.

1 Dissertation Research

My dissertation research focuses on the memory system, which plays an important role in a program's lifetime and contains a wealth of information. The memory system holds a significant fraction of a program's execution state and serves as a medium of communication between the various parts of a program. Current hardware systems export a narrow interface (read and write) and hide most of the data movement operations from software. I believe that future memory systems should be designed in a more programmer-friendly manner. My research proposes hardware mechanisms that shed light on the memory system and expose information that software can use for both self-diagnosis and control of data flow using higher level semantics, such as transactions. Specifically, I developed mechanisms for *Monitoring*, *Isolation*, and *Protection* of memory. These mechanisms have been designed to support fine-grain cache block regions (typically 10s of bytes), which simplifies the interface with program-level objects. Here, I describe the design and use of each mechanism.

1.1 Monitoring

Many program analysis tasks required for debugging, performance optimization, and speculative threading, need to track a program's accesses dynamically and insert instrumentation to record information about the accesses. This introspection is intrusive and computationally expensive — it imposes a 6-30$\times$ performance overhead. My thesis is that hardware support for *monitoring* can help expose the data movement in the memory system to software and reduce this overhead. We can enable monitoring by collecting the coherence events that occur in the shared memory and notifying software about them. I propose two monitoring mechanisms, Alert-On-Update (AOU) and Dependence Summary Counters (DSCs), that differ in when and what information is provided to software.

- *Alert-On-Update (AOU) :* This is a light-weight mechanism in which the hardware triggers a software handler on coherence events and provides information about the event. Software controls the use of and reaction to the event and, can relate the event information to program semantics in various ways.
- *Dependency Summary Counters (DSCs):* While Alert-On-Update provides event-based feedback, often, passively enumerating the events is sufficient. I proposed a set of hardware counters at every processor to enumerate the coherence data transfers with other processors. These counters provide information about shared data and their access patterns.

Using AOU, our group proposed the first hardware-accelerated software-based transactional memory system, which provides the performance of hardware transactions with the policy flexibility of software transactions. I have also used AOU to develop new thread synchronization schemes [TRANSACT'09] and detect atomicity bugs [TR945]. I used the DSCs to concisely track conflicts between transactions [ISCA'08] and to improve thread scheduling, which maps communicating threads to nearby processors [TR945].

1.2 Isolation

Isolation refers to the ability to hide modifications from certain parts of the program and then expose or revoke the changes in bulk. The classical use of isolation has been sandboxing and transactions. In sandboxing, software uses isolation to ensure a faulty plugin doesn't damage the integrity of the main application. Transactions use isolation to ensure that concurrent speculative tasks don't see any intermediate state. Isolation primarily requires support for managing multiple versions of a location, i.e., a new version is needed to buffer the modifications until they are committed while an old version is needed to hold the existing values in case the modifications are discarded.

My hardware mechanism is based on the observation that we can convert the multiple levels of caching in the memory system to hold the different versions. We maintain the new version in the processor's private cache levels and move the old version to the shared cache level. This scheme supports low overhead commit and revocation of isolated data. The overall design is independent of the memory system architecture and I have ported it to snoopy [ISCA'07] and directory protocols [ISCA'08]. To be practically feasible, isolation needs to support an unbounded number of locations — I have investigated software instrumentation [ISCA'07], hardware controller [ISCA'08], and intermediate hardware-software approaches [JPDC'10]

A noteworthy feature of my design is that it permits multiple new versions of a location, allowing different software tasks to isolate the same location concurrently. Our group used this feature to demonstrate that a hardware-based optimistic transactional memory system can be realized within a traditional memory system framework, without any centralized arbitration.

1.3 Protection

Protection seeks to help programmers develop safe and robust applications. Contemporary programs consist of millions of lines of code written by varied sets of developers. Since all the software modules are linked into the same application space, a faulty module makes the entire application vulnerable. I have developed *Sentry*, a hardware framework that allows software to enforce protection between a program's modules [ISCA'10, under submission]. The application only needs to define the policy; Sentry transparently ensures the integrity of a module's private data (no external accesses), the safety of inter-module shared data (enforce the permissions specified), and adherence to the module's interface (regulate function invocation). The key hardware innovation is a light-weight permissions cache that checks accesses after the operating system page protection scheme. It intercepts L1 cache misses and reuses the coherence states to implicitly validate L1 cache hits. From the software's perspective, Sentry appears as a pluggable access control mechanism that can be used to set up various protection policies between an application's modules without requiring operating system intervention during normal operation.

I used Sentry to implement protection for the popular Apache web server and guard the core application against runtime modules. This was achieved with minimal annotations added incrementally to the source code, and minimal performance overheads ($\simeq$10%). I also developed a watchpoint-based tool that could detect a variety of memory-related bugs (e.g., buffer overflow) to demonstrate Sentry's ability to reduce debugging overhead.

2 Future Work

Power and complexity concerns will govern how the transistor bounty will be used in the future. It appears that there will be two distinct paradigms: multicore chips that consolidate many threads and system-on-chip designs that consolidate varied low-cost devices, such as graphics accelerators and I/O, on to the processor's package. Coinciding with this trend is the emergence of new classes of computing. Low-cost mobile devices powered by system-on-chip designs are able to satisfy user needs and enable new applications. Data centers are exploiting multicores to increase compute density and provide remote software services that mirror many desktop applications. My overall future goal is to research the hardware support needed to enable introspection on these platforms and to improve performance and energy efficiency.

2.1 Performance Introspection

In the short term, I would like to use the mechanisms I have developed for monitoring to target performance challenges. Modern processors include many hardware components for performance optimizations, multiple levels of caching, and concurrent threads. The interaction of different hardware optimizations and resource sharing by multiple threads introduces significant performance variability (and often degradation). Information about the system condition and interaction between the various components can help us address these challenges. For example, the DSC and AOU mechanisms specifically provide details about data sharing and cache misses that can be used to correlate hardware activity with program semantics. I have already demonstrated that the information in DSCs can be used to reduce latency of access to shared data by assigning threads that communicate with each other to nearby cores [TR945]. Similarly, we can use the information to guide the management of shared resources such as last-level cache space, cache eviction policies, on-chip network bandwidth, and memory controller scheduling. I will also address the challenge of developing an agent that will coordinate the hardware resources. For tasks like prefetching, a software agent would probably do a better job of interpreting program semantics, while for cache-space management, software would probably be too heavyweight.

Another set of performance challenges are introduced by the increasingly heterogeneous set of devices appearing in computing platforms. The devices integrated on a single chip vary significantly in their functionality (i.e., I/O vs Computation) and have different capabilities (i.e., different instruction set, execution units). Unlike multiprocessors which primarily interact through memory, these devices have varied non-coherent memory models and interact through custom interfaces. I plan to develop hardware mechanisms that shed light on data transfers and computation invocation between these devices. This will help software understand computation spread across these devices, and appropriately manage the resource utilization.

Finally, heterogeneous hardware allows the system to offload the monitoring software in cases where the primary platform has limited capability. Mobile devices suffer from performance, memory, and energy limitations, which dictate the type of analysis tools that can run concurrently with the application. I plan to study the possibility of using a more capable external hardware platform to augment the mobile device. In this environment, AOU will provide information about accesses on the mobile device and the software runtime analyzer will run on a desktop. There are many research challenges that need to be addressed to enable the mobile device to interact with the external device. I will explore the network interface and runtime software support needed to transfer information available at the monitoring mechanism to the remote machine. I will also develop mechanisms to enable the software on the remote machine to react and control the main application.

2.2 Energy-Aware Software Design

In the long term, I see energy efficiency as a formidable challenge encountered by all computing platforms. In data centers, the energy consumed by a server limits compute density and scalability. In mobile devices, there is a cap on the total energy available from the battery, which impacts end-user experience. Typically, system designers have treated energy as a resource issue and dealt with it at the hardware and system software level. Unfortunately, there are many examples of wasteful activity at the application level — screensavers increase power by 20-30% compared to a monitor that is shut off. Furthermore, in mobile devices, the hard cap on the total energy available means that a misbehaving application can impact the energy available for other applications.

I believe it will be beneficial to tackle energy efficiency directly at the source, the application, and increase awareness amongst software developers. We have an opportunity to optimize for energy by enabling software developers to directly observe and react to the energy consumption in applications. This requires exchanging information across the various levels in the system hierarchy. There are two directions in which the information needs to flow. First, hardware needs to help software understand its energy use. Second, applications need to relay some of the program semantics (e.g., idle periods) to hardware controllers, which can configure architectural parameters to improve efficiency.

To expose the energy consumption of architectural components, hardware needs to provide energy counters that directly measure voltage and current loads, and enumerate activity. Current chips have exposed performance counters from which we can extrapolate the power consumption for processor components with reasonable accuracy. We would also need to expose the energy expended by data transfers in the

memory system, interconnects, and I/O interfaces, which is becoming increasingly significant. Another challenge is that a framework is needed for programmers to reason about energy. I plan to relate energy to the highest level of abstraction, the asymptotic complexity of the algorithms. The challenge is we would need to construct energy models that relate algorithmic steps with device energy (which is hardware dependent). In such cases it would be interesting to relate the asymptotically fastest algorithm with the most energy efficient.

Finally, a key aspect of designing energy-aware applications is to permit the software to relay semantic information to the runtime system, which can effectively optimize resource management. I see two promising directions:

- *Energy-aware System Interface :* I envision a system in which the programmer will provide hints to the hardware about computation that might not be needed and make fundamental tradeoffs in the algorithms. Many applications we use every day, such as web search, use an iterative method to successively refine the solution. In this case, the programmer could build in runtime adaption to tradeoff accuracy and iterations, based on the energy available. Similarly, when rendering a video the media player could dynamically change the screen resolution to save energy. An interesting algorithmic tradeoff is using memoization to save and look up solutions, rather than redoing the computation.
- *Energy-Optimal Accelerators :* Increasing transistor budgets have led to peripheral devices that are more energy efficient and faster than general purpose processors for important programming patterns. Based on programmer annotations which indicate the existence of such patterns; the system can optimize for energy by generating code to map the computation to the appropriate device. At the hardware level, I will explore memory system design, on-chip networks, and task queues to interface the accelerators with the general purpose processors. At the software level, I plan to develop runtime libraries that can effectively configure and manage these resources.

In conclusion, I believe that the incorporation of techniques to exchange information between hardware and software will enable us to realize applications that can adapt and address various performance, energy and reliability challenges.