Reflex: Using Low-Power Processors in Smartphones without Knowing Them

Felix Xiaozhu Lin, Zhen Wang, Robert LiKamWa, and Lin Zhong
Rice University
{xzl, zhen.wang, roblkw, lzhong}@rice.edu

Abstract

To accomplish frequent, simple tasks with high efficiency, it is necessary to leverage low-power, microcontroller-like processors that are increasingly available on mobile systems. However, existing solutions require developers to directly program the low-power processors and carefully manage inter-processor communication. We present Reflex, a suite of compiler and runtime techniques that significantly lower the barrier for developers to leverage such low-power processors. The heart of Reflex is a software Distributed Shared Memory (DSM) that enables shared memory objects with release consistency among code running on loosely coupled processors. In order to achieve high energy efficiency without sacrificing performance much, the Reflex DSM leverages (*i*) extreme architectural asymmetry between low-power processors and powerful central processors, (*ii*) aggressive compile-time optimization, and (*iii*) a minimalist runtime that supports efficient message passing and event-driven execution. We report a complete realization of Reflex that runs on a TI OMAP4430-based development platform as well as on a custom tri-processor mobile platform. Using smartphone sensing applications reported in recent literature, we show that Reflex supports a programming style very close to contemporary smartphone programming. Compared to message passing, the Reflex DSM greatly reduces efforts in programming heterogeneous smartphones, eliminating up to 38% of the source lines of application code. Compared to running the same applications on existing smartphones, Reflex reduces the average system power consumption by up to 81%.

Categories and Subject Descriptors D.4.2 [**Operating Systems**]: Storage Management – Distributed Memories; D.4.7 [**Operating Systems**]: Organization and Design

General Terms Design, Experimentation

Keywords Heterogeneous systems, energy-efficiency, mobile systems, distributed shared memory

1. Introduction

An emerging, important category of smartphone applications intend to serve their users in the background by sensing the physical world continuously, e.g., health monitoring, context-aware services, and participatory sensing. They, however, can significantly reduce battery lifetime. The key reason is that existing smartphones only expose their powerful central processors to general-purpose processing, including simple, frequent sensor data processing. Many researchers have shown that it is necessary to add a low-power processor to process simple, frequent tasks [1, 29, 34]. Not surprisingly, emerging mobile Application Processors are including low-power general-purpose cores, e.g., TI OMAP4/5 [37, 38], NVIDIA Tegra 2 [26], and Samsung Exynos 4210 [32]. Moreover, sensors intended for mobile devices are also embracing low-power microcontrollers for in-sensor data processing [27, 36]. The resulting hardware architecture is *asymmetric* with multiple, memory-incoherent cores that differ in processing power by orders of magnitudes. In this work, we refer to a traditional powerful central processor, e.g., Cortex-A9 on OMAP4, as a *central processor* and an added low-power processor, e.g., Cortex-M3 on OMAP4 as a *peripheral processor*. Both central and peripheral processors can be multicore themselves.

Although special APIs and toolkits are available for using peripheral processors, they require developers to not only explicitly treat the resulting system as heterogeneous and distributed, but also explicitly handle inter-processor communication. Not surprisingly, there are very few third-party smartphone applications that actually leverage peripheral processors to save energy.

Our goal is to relieve developers from directly dealing with the heterogeneous, distributed nature of the architecture so that they can easily port legacy applications and develop new ones to leverage peripheral processors for energy efficiency. While this goal is reminiscent of that of programming other heterogeneous, distributed systems, we face a set of unique challenges when dealing with smartphones: extreme architectural asymmetry, energy efficiency, and entrenched operating systems and programming models. Our solution to these challenges is a suite of compiler and runtime techniques, called *Reflex*. With Reflex, a developer only needs to encapsulate the code for a peripheral processor as a special class and limit access to system services in the code. Transformed by the Reflex compiler and assisted by the Reflex runtime, the encapsulated code will be able to run on a peripheral processor with native efficiency and communicate with the rest of the application as if it were also running on a central processor. Reflex is compatible with mainstream smartphone operating systems. Over the past two years, Reflex has been evolving and its evolving path is documented in a technical report [16].

The heart of Reflex is a novel software Distributed Shared Memory (DSM) design. The DSM allows code on multiple heterogeneous processors to easily exchange data, even without a hardware-coherent memory. While numerous software DSM designs have been in use, the Reflex DSM design is unique in its focus on energy efficiency and its support of extreme architectural asymmetry. The Reflex DSM contains multiple technical innovations. First, it maintains architectural energy efficiency by using a peripheral processor to serve coherence requests in order to keep more power-hungry ones in sleep. Second, it creates a shared memory abstraction across processors that differ in processing power by orders of magnitude, and minimizes the disruption in frequent data processing on the peripheral processors. Finally, it

ASPLOS'12, March 3–7, 2012, London, England, UK.

employs code instrumentation and aggressive compile-time optimization to greatly improve the DSM performance.

We report a complete implementation of Reflex and its evaluation using smartphone sensing applications that are classic or reported in recent literature. With minor code changes, Reflex works for a TI OMAP4430 development platform as well as a tri-processor mobile platform built with a commercial smartphone and two microcontrollers. We show that Reflex reduces the average power consumption of the applications by up to 81% compared to the same applications running on existing smartphones, where only the central processors are leveraged. The experimental evaluation also highlights the great development ease provided by a shared memory: compared to the message passing paradigm, the DSM reduces the source lines of application code by up to 38%, with less than 5% source code difference from the legacy counterpart. Moreover, the Reflex DSM incurs low overhead even in a 16-bit microcontroller, i.e., 25% of ROM, 2.5% of RAM, and 3% of processing time.

The rest of the paper is organized as follows. We provide the background and overview of Reflex in Sections 2 and 3, respectively. In Sections 4 to 6, we present the design of the Reflex runtime, DSM, and compiler support, respectively. We provide the prototype realization in Section 7 and report evaluation in Section 8. We address related work, discuss how Reflex can be extended, and conclude in Sections 9 to 11, respectively.

2. Background

We next provide the motivations to Reflex and outline the key assumptions made by the Reflex design.

2.1 Smartphone Sensing Applications

Smartphones and similar devices are embracing a variety of sensors. An important, emerging category of applications often continuously use these sensors without user engagement, such as participatory sensing, context-aware and mHealth applications. However, such applications are known to be power-hungry even with very efficient sensors. For example, the accelerometer inside the Nokia N900 consumes less than 1mW itself when active, yet the entire smartphone will consume 50–100mW of power when simply reading the accelerometer at 30Hz, reducing the battery lifetime to barely acceptable.

The key reason for this inefficiency is that current smartphones only expose their powerful central processors to third-party programming. It is well-known that a lightly utilized powerful processor is inefficient [10], because its advanced architectural features, e.g., deep pipeline, can barely benefit light workloads, but introduce high static power consumption and high overhead for power management. On the other hand, frequent tasks featured in sensing applications are typically very light workloads because developers often employ a cascaded design that conditions the execution of a complicated task on the output of a simple task in order to improve efficiency [19, 28].

2.2 Lower-Power Processors in Mobile Systems

Many have proposed to add low-power peripheral processors to mobile devices [1, 29, 34, 42] and to wireless sensor nodes [21, 24] in order to process simple, frequent tasks. Mobile Application Processors are actually embracing low-power cores [26, 30, 32, 37, 38]. Moreover, sensors intended for mobile devices are also embracing low-power microcontrollers for in-sensor data processing [27, 36]. All result in extremely asymmetric hardware architecture with the peripheral processors orders of magnitudes weaker and lower-power than the central processors. This extreme

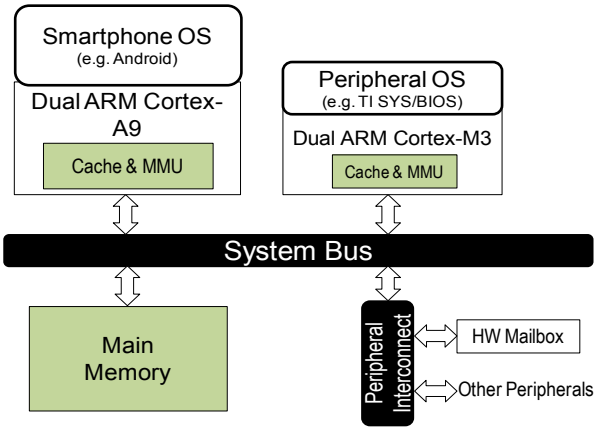


Figure 1. *A TI OMAP4 mobile Application Processor includes two microcontroller cores based on ARM Cortex-M3 in addition to the usual, dual-core ARM Cortex-A9. The M3 cores run their own real-time OS and do not have a coherent memory view with the A9 cores.*

architectural asymmetry is the key to the energy efficiency of accomplishing a wide range of workloads. It also has the following consequences that drive the design of Reflex.

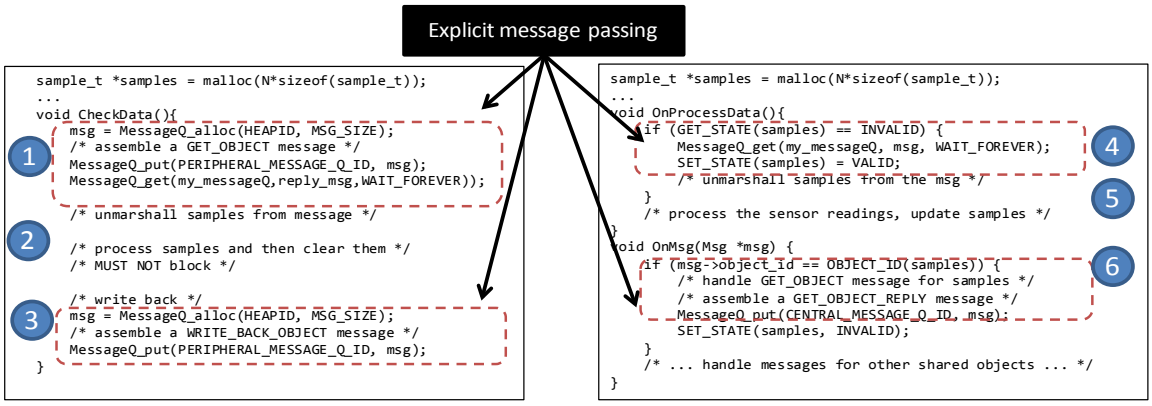
Multiple Microarchitectures*:* A peripheral processor may have a different microarchitecture than a central processor. This makes solutions relying on a single ISA inadequate. For example, an ultra-low-power microcontroller core like TI MSP430 is necessary in achieving mWatt power consumption for continuous sensing on smartphones [29]. TI OMAP4 employs ARM Cortex-A9 as the central processor but ARM Cortex-M3 as the newly added microcontroller processor. Compared to the Cortex-A9 cores, the Cortex-M3 cores only support a small subset of instructions while lacking many advanced features such as SIMD extensions.

Multiple Kernels: The extreme architectural asymmetry also makes a monolithic kernel impossible to manage all processors. While the central processor usually runs a smartphone OS like Android, peripheral processors often run separate kernels optimized for them. For example, an OMAP4-based system runs SYS/BIOS [39] on its Cortex-M3 cores.

Lack of Hardware-Coherent Memory: Hardware coherence is difficult for smartphones with extremely asymmetric processors. Peripheral processors can barely realize a hardware coherence protocol under tight energy constraint. Furthermore, the microarchitectural difference between strong and weak cores makes hardware coherency even less attractive. For instance, TI OMAP4 Cortex-A9 and Cortex-M3 cores physically share non-coherent memory. To exchange data, the cores of different types need to copy data to and from the memory, and notify each other with the hardware mailbox shown in Figure 1.

2.3 Programming Challenge

The system resulting from extreme architectural symmetry is heterogeneous and distributed. To leverage peripheral processors, a developer not only needs to learn a new programming environment but also needs to carefully manage inter-processor data exchange. This is because simple, frequent tasks of sensor data processing in a sensing application often need to exchange data with the rest of the application. Examples of such data include sensing parameters (static or adaptive), intermediate or final results, and control/status variables. Such data can be read or written by multiple code parts. Furthermore, the sensor data processing



(a) The code for the ARM Cortex-A9 central processor (b) The code for the ARM Cortex-M3 peripheral processor

Figure 2. *Example code based on message passing. The code is written for TI OMAP4 based on the Syslink inter-processor message passing API. To make the list concise, we hide low-level code such as message queue management.*

```
lock m; sample_t *samples; /* global variables */
...
samples = malloc(N*sizeof(sample_t));
...
void CheckData(){
    acquire(m);
    /* process samples and then clear them */
    release(m);
}
```

(a) The code for the ARM Cotex-A9 central processor

```
void OnProcessData(){
    acquire(m);
    /* process the sensor readings, update samples */
    release(m);
}
```

(b) The code for the ARM Cortex-M3 peripheral processor

Figure 3. *Example code based on shared memory. A lock is used to prevent data race.*

task accesses the shared data more frequently than its complicated, infrequent counterpart, often by orders of magnitude, due to the cascaded design mentioned in Section 2.1.

Without hardware coherent memory, all existing programming solutions support data exchange with certain forms of message passing. With systems reported in [1, 34], the developer has to explicitly coordinate the data exchange between central and peripheral processors. To exchange data between a Cortex-A9 core and a Cortex-M3 core on the TI OMAP4, the developer must use an inter-processor message passing API called Syslink provided by TI. Not surprisingly, there are very few third-party smartphone applications that actually use peripheral processors to save energy.

2.3.1 Problems with Message Passing

Message passing is a programming paradigm where distributed code parts communicate by explicitly copying messages, being data or program objects. Programming smartphone sensing with message passing is a nontrivial task. We next use an example based on OMAP4 to illustrate the reasons. Figure 2 shows two code snippets excerpted from a simple sensing application, which uses messages to exchange data with the Syslink API. The two code snippets are executed on two different processors, and both use the `samples` array. The peripheral processor code, running on a Cortex-M3 core, fills `samples` as it acquires sensor readings, and the central processor code, running on a Cortex-A9 core, reads `samples` and clears historical values when necessary.

In the code, message passing exhibits its well-known drawbacks. First, developers need to synchronize distributed copies of the same object with a customized communication protocol, (1) (3) (4) (6) in Figure 2. Due to the discrepancy of ISAs, developers may need to convert `samples` among different data formats. For shared objects containing pointers, e.g., a linked list, the efforts will be more tedious.

The use of message passing is further complicated by the extreme architectural asymmetry. Message-based communication should not interfere with the sensor data processing on the weak, peripheral processors. For example, the computation in (2) should complete in time, because the peripheral processor is stalling its frequent processing task (5) and waiting. Similarly, message handling (6) on the peripheral processor should not be executed so often in order to avoid delaying the frequent data processing. Finally, for energy efficiency, the developer must arrange the communication so that no message should wake up the central processor from a sleep state.

2.3.2 Problems with Software DSM

The archrival to message passing in programming distributed systems without hardware-coherent memory is software DSM. With a software protocol that coordinates multiple distributed memory components to collaboratively serve memory access, software DSM creates the key illusion of a shared address space. Compared to messages, shared memory is a much more familiar abstraction to smartphone developers. The resulting application code is concise and similar to legacy smartphone code, as shown in Figure 3. However, DSM may introduce communication overhead due to thrashing or unnecessary data transfer.

Reflex resorts to DSM in order to conceal the underpinning messages, with the following key rationales. First, compiler and runtime can automate the architecture-dependent mechanics of message passing. Second, overhead of DSM diminishes as access contention decreases [14]. Fortunately, sensing applications tend to have light access contention, since only the data processing task is frequently accessing shared objects. Finally, with compiler techniques, the overhead can be reduced close to what can be achieved by hand-optimized message passing.

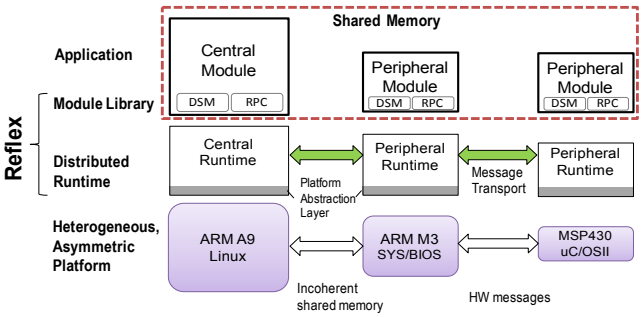


Figure 4. *The structure of Reflex. This example heterogeneous, asymmetric hardware platform is a TI OMAP4-based smartphone with an added MSP430 processor.*

3. Overview of Reflex

Our goal in developing Reflex is to relieve developers from directly dealing with the heterogeneous, distributed platform so that they can easily port legacy applications and develop new ones to leverage the peripheral processors for energy efficiency.

3.1 Programming Constraints

In order to make the system design practical, we require application developers to be aware of peripheral processors in the following two ways. First, developers need to properly encapsulate the code for a peripheral processor as a *peripheral module*. Accordingly, we call the code designated for a central processor a *central module*. To ease the development of peripheral modules, Reflex provides a virtual base class called *ModuleBase*. Developers write a peripheral module as a subclass of ModuleBase.

Second, Reflex restricts system services that a peripheral module can use, since a peripheral processor does not have all the resources available on a central processor, e.g., Internet connectivity. Besides, sensor data processing, according to our observation, only requires limited system services: dynamic memory, timer, and sensor data acquisition.

3.2 How Reflex Works

With an application properly developed as discussed above, the Reflex compiler, the distributed runtime, and the library collectively create an illusion that the application is running on a single processor. The key to this illusion is to support data exchange among modules running on different processors. Reflex recognizes two forms of data exchange: procedure call and shared memory. Reflex automatically translates a procedure call between distributed modules into a synchronous remote procedure call (RPC) and supports the shared memory with software DSM. While the RPC realization is standard and straightforward, the DSM poses significant technical challenges due to the extreme architectural asymmetry.

When the application is launched, Reflex treats the central module as a normal smartphone application and executes it on the central processor. Then, the runtime ships the peripheral module binaries to the proper peripheral processors for execution. When the central module terminates, the Reflex runtime will terminate all running peripheral modules of the application.

3.3 Hardware Requirements

We design Reflex to have minimalist hardware requirements for peripheral processors and their integration with the central processor: at least 8KB of ROM, several KB of RAM, the capability to handle interrupts, and the capability to acquire sensor data without involving the central processor. Reflex only requires a data link capable of interrupt-driven and bi-directional data exchange between peripheral and central processors. The data link can be a low-speed hardware bus, e.g., I^2C; it can also be realized with a physically shared memory and hardware support for inter-processor interrupts, as in TI OMAP4.

Reflex does not require a hardware Memory Management Unit (MMU), although many existing distributed systems leverage MMU to create a shared address space or to trap local memory operations. Despite some peripheral processors indeed have dedicated MMUs, e.g., Cortex-M3 on OMAP4, it is very difficult to implement MMU for even weaker peripheral processors that are necessary for energy efficient sensing applications.

4. Distributed Runtime Design

Figure 4 shows the structure of Reflex, including the distributed runtime and the library. This user-level implementation makes Reflex compatible with existing smartphone operating systems and platforms, although a more disruptive kernel-level implementation is possible.

As shown in Figure 4, the Reflex runtime has one component on each processor: the *central runtime* on the central processor as the coordinator and the *peripheral runtimes* running on the peripheral processors. Locally, a component runtime provides the corresponding module with a unified abstraction of resources. We next describe in detail two important services provided by the distributed runtime.

4.1 Module Execution Model

Unlike many other heterogeneous systems that execute offloaded code synchronously, Reflex executes modules in parallel. To accommodate the DSM and the RPC, the Reflex runtime is enhanced to execute modules with an *event-driven* model, which is implemented with native execution units (e.g., thread or process).

A Reflex event is a formatted data unit sent to a module from either other modules or the local component runtime. These data units can be inter-processor messages (i.e., the primitive used to implement the DSM and the RPC) or can be used in providing local system services. A module has a private event queue to which the local component runtime asynchronously delivers events.

A module must *poll* its event queue to retrieve an event. Polling is non-blocking and lightweight: when the event queue is empty, it only involves checking a flag variable. Such a polling-based model provides the module with the control over exactly when to handle asynchronous events. Polling occurs 1) when the module finishes executing an event handler, its control flow returning into the event loop and 2) according to the instrumentation by the Reflex compiler. In particular, when a module is waiting to receive some DSM or RPC message and stalls the execution of its application code, it also polls its event queue to process DSM and RPC messages that are received but not yet handled. Compared with preemptive event handling, the polling-based model prevents data processing on a weak processor from being overwhelmed by coherence requests sent by stronger processors.

With the event-driven execution model, the code of a module consists of three parts: the event loop as the skeleton of the module, which keeps retrieving and dispatching various events; the application-specific handling of events, as provided by the developer; and functions that support the DSM and the RPC as well as wrapping the system services.

4.2 Message Transport

Globally, all runtime components collectively implement a lightweight message transport as shown in Figure 4. The message transport serves as the only interface for communication among all runtime components as well as the lowest-level primitive of communication among all distributed application modules, a design also adopted by many other heterogeneous systems, e.g., Helios [25] and Barrelfish [3]. A message will be delivered as an event to the destination module.

The message transport is best-effort and does not guarantee reliable delivery. This is because higher-level protocols on top of the message transport, such as the DSM and the RPC, all use messages in request/response pairs between the two communicating modules. Thus, the response message implicitly indicates the delivery of the earlier request message. Furthermore, the higher-level protocols employ message sequence numbers to identify and therefore discard duplicate messages due to timeout and retransmission.

5. Software DSM Design

We now present the design of the Reflex software DSM. The key design objectives are to leverage the architectural asymmetry for energy efficiency while minimizing performance overhead.

5.1 General Design Choices

Overall, the Reflex DSM realizes release consistency and maintains memory coherence at object-level, as will be discussed below.

Memory Consistency Model*:* To developers, the memory consistency model defines the expected outcome of memory accesses. The Reflex DSM implements release consistency [35]. Release consistency introduces two synchronization operations (`acquire` and `release`) and guarantees that a *correctly* synchronized application will exhibit sequential memory consistency, just as if the application were running on a single processor. By *correctly*, we mean using synchronization operations (`acquire` and then `release`) to construct critical sections to prevent concurrent accesses to the same object, if at least one of these accesses is write.

We choose release consistency for two reasons. First, it is easy for smartphone developers to use. We notice that writing synchronized applications is already a common task in contemporary smartphone development, e.g., Android application development in Java greatly encourages synchronized applications [22]. Second, release consistency is lightweight to implement: it can greatly reduce communication overhead by allowing most communication to happen at synchronization points, a significant benefit to our targeted architecture where the inter-processor communication latency is relatively high.

Coherence Granularity: A DSM design must determine the basic memory unit for which coherence is maintained, i.e., the coherence granularity. A larger unit may better leverage spatial locality to amortize communication overhead, but also increases unnecessary data transfer due to false sharing.

Many DSM solutions support *block-level* granularity, using a fixed-size memory block (or a fixed-size page defined by the MMU) as the basic memory unit for coherence. In contrast, the Reflex DSM uses software objects as the basic memory unit for two reasons. First, Reflex does not assume an MMU. Second, a pre-defined block size leads to a fixed tradeoff between the communication overhead and the false sharing of the resulting DSM. In contrast, *object-level* granularity allows the Reflex DSM to employ compiler-supplied program information to better leverage locality.

The Reflex DSM supports sharing two types of objects: global objects and heap objects. In facing the extremely limited local memory on peripheral processors, a module only caches heap objects it accesses, rather than all heap objects in the application.

The Reflex DSM associates each shared object with an application-wide integer ID that is valid across all modules. The IDs are allocated in ascending order as objects are created, either statically by the compiler (for global objects) or dynamically by the DSM (for heap objects). Each module has its own range to allocate new IDs so it can do so independently. The DSM maintains a table of shared objects in every module; each entry in the table records the metadata of one shared object, including the object ID, type, its local address range, and information about other modules that also share this object.

We next present the *coherence protocol* based on the design choices made above. The protocol specifies how modules sharing an object should behave in accessing the object.

5.2 Protocol Invariant

At the core of our coherence protocol is the following invariant:

Any coherence communication between two processors must be initiated by the stronger one.

This invariant guarantees the energy efficiency of the Reflex DSM: in accessing shared objects, a weak processor will never wake up stronger processors with coherence communication. Such a unique invariant clearly distinguishes our coherence protocol from existing ones, as will be described in detail below.

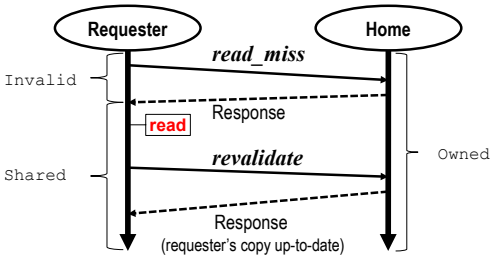
5.3 Asymmetric Module Roles

Given a shared memory object in an application, we define its *sharing group* as all modules in the application that access it. Each module in a sharing group has a local copy of the shared object. All write/read operations by a module on a shared object are actually performed on the local copy of the object.

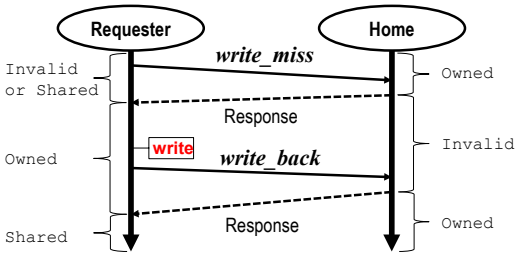
In a sharing group, all modules have asymmetric *roles* based on the relative processing power of the processors that these modules are executed with. To facilitate the discussion, we mention a module as strong or weak to refer to the relative processing power of its associated processor. The weakest module in the group is the *home* of the object, while the other modules are *requesters* of the object. Since a module may access multiple shared objects, it can participate in multiple sharing groups and play different roles in them.

While the home of a global object is assigned by the compiler statically, the home of a heap object is determined by the DSM on-the-fly and automatically. Initially, a heap object *O* allocated by a module *M* has *M* as its home; later, if a pointer to *O* is passed to another module *N* that is weaker than *M*, *N* notifies *M* that itself will be the new home of *O* and retrieves the metadata of *O* from *M*. After that, *M* redirects any subsequent requester of *O* to the new home *N*, a one-time effort for each requester.

The role asymmetry employed by the Reflex DSM is different from existing heterogeneous DSM solutions in an important way. With performance as the primary goal, most existing heterogeneous DSM solutions employ the most resource-rich (and power-hungry) processor to serve requesters [9, 41]. In contrast, the Reflex DSM greatly favors energy efficiency and, therefore, uses the weakest processor to host the home role. Only if the weakest processor handles memory requests will the protocol invariant (Section 5.2) hold. Therefore, other stronger processors are able to remain in sleep mode as much as possible.



(a) *read_miss* and then *revalidate* from a requester



(b) *write_miss* and then *write_back* from a requester

Figure 5. *State transitions in the Reflex DSM coherence protocol. All transitions are triggered by requests from a requester.*

5.4 Finite-state Machine Specification

We now provide details regarding the behaviors of the home and requesters in terms of finite-state machines. A module has a state with regard to any sharing group it participates in; if participating in multiple sharing groups, it will have multiple independent states with regard to each of those groups. A module state can be one of the following two:

- The home can be either `Owned` or `Invalid`.
- A requester can be `Owned`, `Invalid`, or `Shared`.

The states have the same semantics as in most coherence protocols. In a sharing group, conceptually one and only one module is in the `Owned` state. A module in the `Owned` state can read and write to its local copy; a module in the `Invalid` state can neither read nor write to its local copy; a requester module in the `Shared` state can read but not write to its local copy. A module in the `Owned` or `Shared` state has the most updated value of the shared object in its local copy.

Modules in a sharing group change states only by requests sent by a requester. The home module never initiates a change itself. A requester module uses four requests: *write_miss*, *write_back*, *read_miss*, and *revalidate*.

Figure 5 (a) and (b) illustrate the interactions between a requester and the home. In the two cases, the requester needs to read and write to the object, respectively. Before reading the object (Figure 5 (a)), the requester in the `Invalid` state sends a *read_miss* request to the home. The home responds by sending back the most updated value of the shared object. The *read_miss* request will change the state of the requester to `Shared`. Later, when the requester sends a *revalidate* request to the home, the home responds to indicate that the object has not been modified since the last *read_miss* request. Therefore, the requester module remains in the `Shared` state. If the home responds that the object has been modified (not shown in the figure), the requester will transit to the `Invalid` state, and it will need to send a *read_miss* again prior to any further read of the object. Note that *revalidate* is only necessary during synchronization, as we will explain in detail in Section 5.6.

Before writing to the object (Figure 5 (b)), the requester in the `Shared` or `Invalid` state sends a *write_miss* request to the home in order to transit into the `Owned` state; it sends a *write_back* request (along with the latest value from the local copy) to the home in order to transit back into the `Shared` state. Accordingly, the home will transit to `Invalid` and `Owned` after responding to *write_miss* and *write_back* requests, respectively.

The protocol described above introduces two types of latency in applications, namely *home latency* and *requester latency*. Home latency is introduced when a home in the `Invalid` state stalls its execution and passively waits until it is in the `Owned` state. Requester latency is introduced when a requester stalls its execution to wait for responses to *read_miss* or *write_miss* from the home. We will experimentally evaluate the impact of the above latency to applications in Section 8.

5.5 Support for Synchronization

Release consistency defines two synchronization operations for applications: `acquire` and `release`. The Reflex DSM provides *lock* objects to support both operations, implementing a lock as a shared object using the coherence protocol. For example, to `acquire` or `release` a lock, a requester sends the home of the lock object a *write_miss* or *write_back* request of the lock, respectively. To ensure release consistency, Reflex currently requires a lock to have the same home as its protected objects have. We plan to remove the restriction in future work.

According to release consistency, if a module performs `release` and another module subsequently performs `acquire`, the first module must make its updates visible to the second one. In propagating updates during synchronization, homes are *lazy* and requesters are *eager*. When the home `releases` a lock, it performs no coherence communication; it will send an updated value to a requester only when the requester asks for it. In contrast, when a requester `releases` a lock, it writes back all its updated objects that have the same home as the lock has.

Such asymmetric module behaviors are critical to the design goals of the Reflex DSM. The home's laziness guarantees high energy efficiency, as it maintains the protocol invariant stated in Section 5.2. At the same time, the requester's eagerness keeps the home's stalling period small in order to minimize the resulting disruption to frequent processing.

5.6 Notable Design Aspects

We next highlight three notable design aspects that are essential to the objectives of the Reflex DSM. First, a home resolves its access of an unavailable object by passive waiting. This is to maintain the protocol invariant: running on the weakest processor, the home cannot initiate communication to other modules.

Second, to keep an object updated, a requester in the `Shared` state actively revalidates the object with the home. This is unlike most other DSMs where a read-only object is passively invalidated or updated by other hosts. The reason is that the home cannot send out messages to invalidate or update requester's copies. Instead, the home tracks whether a `Shared` requester's copy is still up-to-date, by observing whether the object has been written since the requester's last *read_miss* or *write_back* request. Accordingly, the home responds to *revalidate* requests, as described in Section 5.4.

One may think that the requester-initiated revalidation incurs excessive communication, but actually it does not. In a nutshell, on `acquire` of a lock, a requester will revalidate all its objects that

have the same home as the lock has, rather than sending a revalidation request before each read access. This is because release consistency guarantees that a module sees updated values only after each `acquire` operation. In addition, the Reflex DSM piggybacks the communication of revalidation to that of `acquire`, adding no more than one byte per message.

Third, the Reflex DSM aggressively reduces home latency at the cost of increased requester latency. This is achieved by keeping the home in the `Owned` state as much as possible, with the following designs. 1) In propagating updates during synchronization, requesters are eager and the home is lazy, as described in Section 5.5. 2) The home handles coherence requests only when its frequent processing is idle or stalled, supported by the execution model of modules as discussed in Section 4.1. This is because of the extreme resource asymmetry: requesters, running on stronger processors, can produce requests at a much higher rate than the home, running on the weakest processor, can handle. Furthermore, we choose to mitigate requester latency by compiler optimization described in Section 6 below.

6. Compiler Support for DSM

Without assuming MMU support, the Reflex DSM cannot perform coherence operations within trap handlers, as many other DSM systems do. Instead, the Reflex compiler statically automates invocations to coherence operations in developer's code. In addition, the compiler aggregates nearby coherence operations to greatly reduce the DSM overhead.

6.1 Code Instrumentation

In order to realize the coherence protocol specified above, the Reflex compiler analyzes the application code and inserts two types of code: `pre-access` and `post-access`. By design, for each access of a shared object the Reflex compiler inserts necessary `pre-access` code immediately before the access. The `pre-access` code determines the target object of the access and checks the corresponding module state. It does nothing if the state is `Owned` (before read or write access) or `Shared` (before read access). Otherwise, it performs coherence operations that are specific to the nature of the access (read or write) and the role of the module (home or requester). In addition, after a write access in a requester module the Reflex compiler inserts `post-access` code that sends out a *write_back* request and sets the requester back to `Shared`.

6.2 Optimization for Communication

The inserted code introduces overhead in both checking module state and inter-module communication. The Reflex compiler reduces the former overhead by employing a batching technique similar to that of Shasta [33]: if a module state has been checked by an earlier `pre-access` and no related coherence communication has occurred since then, a later `pre-access` can be eliminated. In the following discussion, we will focus on optimizations for reducing the communication overhead.

Batch Prefetching: The Reflex DSM maintains coherence for each shared object. However, fetching each object with an individual message incurs high communication overhead. The Reflex DSM leverages spatial locality to amortize the communication overhead by speculatively fetching multiple objects in a batch.

Once `pre-access` code finds it is necessary to send a *read_miss* or *write_miss* to fetch an object from the home, the DSM also examines objects that have the same home and adjacent IDs. Since object IDs are allocated in ascending order, having adjacent IDs implies potential spatial locality in accessing such objects: they are either global objects defined close to each other in the source code or heap objects allocated consecutively during execution. The DSM adds multiple objects to a batch, fetches them, and receives responses with two single messages, respectively. We will experimentally show the impact of batch parameters in Section 8.

Deferred Write-back: Due to temporal locality, it is common that a shared object is repeatedly accessed in a short period of time, for example, in a loop. By only inserting one `post-access` after all these accesses, the Reflex compiler essentially merges multiple *write_back* requests of the same object, greatly reducing the number of messages.

A requester should defer *write_back* only moderately, in order to keep home latency low. Given that requesters are executed much faster than the home, the DSM leverages release consistency to defer all *write_back*s until the next `release`. The compiler does so by inserting `post-access` right before each `release` to send out all deferred *write_back*s.

To summarize, by design, before each access of shared objects the compiler will insert `pre-access`; however, after optimization only a small, necessary portion of such code is actually emitted. If `pre-access` in requesters finds out that the current module state disallows the memory access, it immediately sends a message of *read_miss* or *write_miss* (with batch-prefetching). In addition, the compiler inserts `post-access` before each call site of `release`. Such code in requesters immediately sends a message of all deferred *write_back*s.

7. Prototype Realization

We have implemented the complete Reflex design in a modular way for portability. In particular, the DSM and the RPC are fully implemented in the module library that is linked into the module code, as shown in Figure 4. The DSM internally uses the message transport supplied by the Reflex runtime. The Reflex compiler instruments the developer's code by inserting invocations of coherence operations defined in the module library. Our Reflex implementation, with minor platform-specific revisions, runs on two platforms, TI OMAP4 and a custom tri-processor platform.

7.1 Reflex Runtime and Library

We realize the Reflex runtime and library with the software structure shown in Figure 4. On the bottom of the runtime is a Platform Abstraction Layer, which wraps all platform-specific functions, such as kernel API and hardware message passing. On top of the Platform Abstraction Layer, we build all major runtime functions and the module library that implements the DSM and the RPC.

The execution model of a module is implemented with the module library, which is dynamically linked with the developer's code during module execution. The module library is implemented with around 1500 lines of commented C/C++ code, consisting of routines that are for the event loop, for the DSM and the RPC, and for wrapping the system services. The Reflex runtime is implemented in around 4000 lines of commented C/C++ code.

7.2 Reflex Compiler Toolkit

We build the Reflex compiler as a transformation pass on top of the LLVM compiler infrastructure [13]. LLVM compiles the source code into LLVM-IR and the Reflex compiler instruments the IR for DSM. Before the module is deployed as part of the application, the module IR is transformed into to a native binary depending on the target processor.

The object-level coherence of the Reflex DSM requires object types to be determined at compile time. To meet such a need, currently Reflex supports application development with a strong-

typed subset of C++, by applying rules such as using typed malloc, disallowing pointer arithmetic, and disallowing type-casting, a strategy used in many embedded systems [7]. After compiling the application source code into LLVM IR, the Reflex compiler certifies that the IR is strong-typed; it then outputs descriptions of all types in Interface Description Language (IDL).

Reflex includes a small stub compiler that produces code for (un)marshaling objects across different ISAs. Implemented with around 700 lines of Python code, the stub compiler takes the compiler-generated type descriptions and produces (un)marshaling code used by the DSM and the RPC. The major task of the (un)marshaling code is to convert the formats of typed objects. Most data conversion is straightforward except for pointers. Since processors have separate address spaces, one pointer in a module needs to be converted to be used by another module. To do so, the (un)marshaling code uses a *portable pointer* as the intermediate format among local pointers in separate address spaces.

A portable pointer has 32 bits (the same as the longest local pointer in the system), encoded with the following information: object ID (13 bits), the byte offset within the object (10 bits), type id (6 bits, as used in the compiler-generated IDL), and the home id (3 bits). Before sending out a local pointer in a message, the marshaling code replaces the local pointer with a portable pointer in place. On receiving a portable pointer in a message, the unmarshaling code retrieves the corresponding local pointer based on the encoded object ID and offset. If the unmarshaling code finds the object ID is new to this module (e.g., a heap object newly created by another module), the code allocates an object on the local heap with the type specified in the portable pointer, and adds a corresponding entry in the local object table.

7.3 Hardware Platforms

The Reflex runtime and module library are highly portable, thanks to the layered structure. After changing the marshalling implementation, the message passing implementation, and hundreds of source lines in the Platform Abstraction Layer, we have ported Reflex to two platforms: TI OMAP4 and a custom, tri-processor hardware platform.

7.3.1 TI OMAP4

We have realized Reflex on TI OMAP4. We built the Platform Abstraction Layer to wrap the Syslink inter-processor message passing API, the Linux kernel, and the SYS/BIOS kernel. In particular, the message passing abstraction offered by Syslink enables us to implement the Reflex DSM on OMAP4, despite the fact that message passing is actually using a non-coherent shared memory. Due to page limit, in the rest of the paper we will focus on the implementation and evaluation using the tri-processor hardware platform, which is introduced for the first time by this paper.

7.3.2 Tri-processor hardware platform

We developed the tri-processor hardware platform by extending a Nokia N900 smartphone. As shown in Figure 6, the platform employs the N900's powerful OMAP3630 CPU (OMAP3) as the central processor and employs two ultra-low-power microcontrollers, LPC1343 (LPC) and MSP430F1611 (MSP), as the peripheral processors. Table 1 summarizes the characteristics of these three processors. The platform uses Maemo Linux shipped with the N900 as the central kernel and runs two separated µC/OS-II [12] on two peripheral processors. Built on top of the Platform Abstraction Layer, the peripheral runtime is a set of procedures

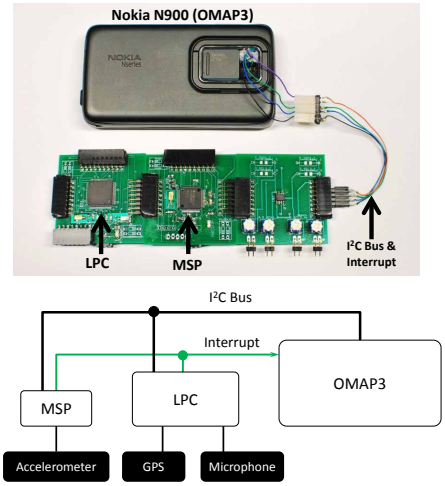


Figure 6. *The tri-processor platform (Top) and its architecture diagram (Bottom)*

Table 1. *The processors used in the tri-processor prototype. At run time, unused processors are turned off to reduce the system idle power.*

	OMAP3	LPC	MSP
Clock Rate	600MHz	72MHz	3MHz
Local Memory	256MB	8KB	10KB
Local Storage	32GB	32KB	55KB
Active Power	~200mW	42.9mW	7.5mW
Idle Power	13.4mW	7mW	3.2mW

linked with the vanilla µC/OS-II kernel. Peripheral modules are executed as µC/OS-II tasks.

I^2C-based interconnect: The three processors are integrated via an I^2C bus at 100KHz and physically share no memory. In order to physically access the I^2C interface and the interrupt line of the OMAP3, we remove the N900's camera module and hijack its connector with the OMAP3. In moving data on the I^2C bus, a byte chunk with a size of typical Reflex messages usually takes tens of milliseconds (6ms for a 48 byte chunk and 18ms for a 176 byte chunk). Built on top of the I^2C bus, the message transport of the Reflex runtime itself incurs little computing overhead, which takes 1500 cycles (MSP), 891 cycles (LPC), and 1800 cycles (OMAP3) in sending a message, and 1560 cycles (MSP), 1612 cycles (LPC), and 1800 cycles (OMAP3) in receiving a message, excluding actual byte sending/receiving. Compared to the tens of milliseconds of interconnect baseline latency, the computing overhead is negligible.

Sensors: The N900, like all commercial mobile devices, tightly integrates its built-in sensors with the central processor (the OMAP3). Therefore, it is very difficult for a peripheral processor to access the built-in sensors without waking up the central processor, violating the hardware requirements of Reflex as outlined in Section 3.3. As illustrated by Figure 6, we add a KXM52 tri-axis accelerometer to MSP through ADC; we add an analog microphone and an MN5010HS GPS receiver to LPC through ADC and UART, respectively.

8. Evaluation

We evaluate Reflex and its DSM with the tri-processor platform reported in Section 7 and four real-world smartphone sensing applications.

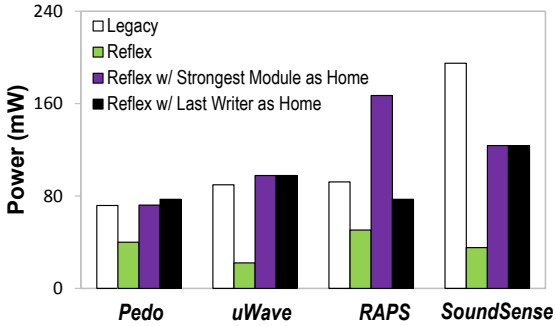


Figure 7. *The average system power consumption of benchmark applications*

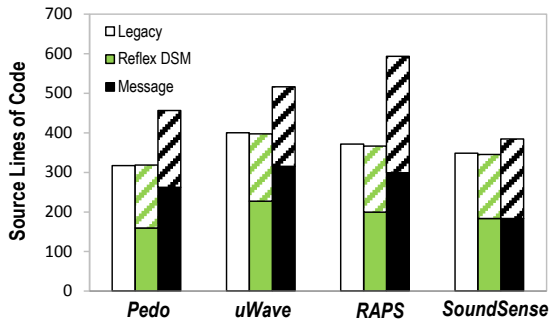


Figure 8. *The source lines of code of benchmark applications. Comment or blank lines are excluded. For a column that consists of two parts, the bottom part shows the source lines of the central module; the top part shows that of the peripheral module(s).*

8.1 Benchmark Applications

Since there is no standard sensing application benchmark, we employ the following four benchmarks, a combination of classic and recently reported smartphone sensing applications. Table 2 provides the typical usage of them used in the reported evaluation.

Pedometer (*Pedo*) uses the accelerometer to count the user steps. *Pedometer* (Reflex) consists of a peripheral module that counts steps and a central module that provides a user interface (UI) to query the step information. The two modules share a few flags and variables that store step count, stride, velocity, distance, and history. The peripheral module can be executed by MSP.

uWave recognizes user gesture with the accelerometer [18]. *uWave* (Reflex) consists of two modules: a peripheral module periodically performs acceleration pattern matching and calls the central module when a gesture is recognized; a central module provides a UI. The two modules share two acceleration templates and a window of recent accelerations, each of which is a 64-element integer array. They also share variables for gesture type, adaptive threshold, and similarity. The peripheral module can be executed by MSP.

Rate-Adaptive Position Sensing (*RAPS*) [28] calculates geo-location uncertainty by comparing current acceleration with context-specific historical averages of velocity and acceleration. When the uncertainty passes a threshold, *RAPS* takes a GPS reading and updates the historical measures. *RAPS* (Reflex) consists of two peripheral modules PA and PB, and the central module. PA monitors acceleration and calculates location uncertainty. When the uncertainty exceeds a threshold, PA calls PB. PB then gathers GPS data, updates the recorded geo-locations, and provides PA with the historical averages. The central module provides a UI to query the recorded geo-locations. PA and PB share information about uncertainty and the historical accelerations. PB and the central module share recent geo-locations with timestamps. PA and PB are executed by the MSP and LPC, respectively.

SoundSense [19] detects sound and analyzes it to determine the mobile user context. *SoundSense* (Reflex) uses a peripheral module to run the frame admission and feature extraction algorithm and a central module to execute simplified context classification. The admission parameters, the window features, and their variances are shared between the two modules. The peripheral module can be executed by the LPC.

We implement each benchmark in three ways: The N900's programming framework (*Legacy*), on the tri-processor platform with message passing (*Message*), and with the Reflex DSM (*Reflex DSM*). A simple UI is implemented for central modules using the Qt framework for N900. Legacy implementations use the N900's built-in sensors while the other two use sensors connected to the peripheral processors. Like most smartphone sensing applications, these benchmark applications spend most time in common scenarios of simple sensor data processing, which determine the overall energy characteristics of the applications.

Table 2. *Benchmark applications used in the evaluation*

Apps	Assumptions of Typical Usage
Pedo	Step detected every 400ms, UI query every 30 sec
uWave	UI activated every 15 min
RAPS	350 new geo-locations per day, GPS on 27% of time [28]
Sound -Sense	Frame sampled every 640ms, 20 events per hour [19]

8.2 Overall Performance of Reflex

We first examine how well Reflex as a whole achieves its goal in saving energy, by measuring the power of the entire system. To do so, we sample current and voltage using USB-2533 (from Measurement Computing) at a measurement rate of 100 HZ. For Legacy implementations, we measure the power of the N900 from its phone-battery hardware interface directly: we physically tap into the interface and simultaneously sample the current (using a 0.1ohm current sense resistor) and the voltage. For the Reflex-based implementations on the tri-processor platform, we additionally include the power of all added hardware. In both cases, we power off unused hardware such as the baseband.

Sensing applications in benchmarks perform periodic processing. We benchmark an application scenario by running it for a given period of time (typically 10 min) with Legacy and Reflex, respectively. During the benchmark period, Legacy and Reflex finish the same amount of sensor data processing. Therefore, the average power over the benchmark period reflects the system energy efficiency.

As expected from the addition of low-power cores, our measurement shows that Reflex reduces the system power consumption by up to 83% for the most common scenarios of each benchmark. Only in scenarios when the central processor is involved do the Reflex-based implementations incur slight power overhead (~3%) from the added peripheral processors. Because such scenarios are relatively rare in real life usage, the Reflex-based implementations will be significantly more efficient than the Legacy implementations. With the per scenario actual power measurement, we emulate the scenario transitions (based on the application usage in Table 2) to compute the application average power. As shown in Figure 7, compared to Legacy implementations, Reflex reduces

the average system power consumption by to up to 81% (65% on average).

8.3 Source Code Examination

We investigate how Reflex facilitates application development by examining the source code of the benchmarks. First, how is Reflex different from the legacy development style? As shown in Figure 8, the Reflex-based and the Legacy implementations of the same benchmark have very similar numbers of source lines of code; more importantly, they share most of the source code with 95% identical source lines. Their source code only differs in the way of gluing the programming framework and invoking the system services described in Section 3. This observation validates that Reflex has achieved its goal of maintaining the contemporary programming style and facilitating porting legacy sensing applications to the heterogeneous, distributed architecture.

Second, how does the DSM ease the application development compared to message passing? To make the source code comparison fair, we hand-optimized the message-based code with best efforts, including implementing the commonly-used message manipulations as subroutines and managing objects with a table and iterating over the table to operate the objects. The DSM greatly eases the application development by enabling data exchange in benchmarks to be concisely coded as in the source code of Figure 3. Compared to the message-passing implementations, the DSM reduces the lines of code by 30% on average as shown in Figure 8. In particular, the synchronization operations required by DSM impose a small development burden: they take at most 10 lines per application.

8.4 DSM Performance

We next zoom into the behavior of the Reflex DSM and examine the effectiveness of its important design choices.

8.4.1 Asymmetric Module Role Assignment

The Reflex DSM exploits the weakest processor to host the home module of a shared object and, therefore, keeps more powerful processors in sleep, a strategy that is critical to the energy efficiency goal. We compare this strategy with the following two alternatives that are widely used in other DSM designs: 1) using the strongest processor for the home module and 2) using the module that last wrote to the object as the home module. Based on the per scenario actual power measurement, we show the emulated overall power consumption under both alternative strategies in Figure 7 ('Strongest Module as Home' and 'Last Writer as Home'). With the alternative strategies, the system power consumption is up to five times as much as that of the Reflex DSM, because the strongest processor is woken up to serve coherence requests and therefore unable to remain in sleep mode for a long time.

8.4.2 Storage and Memory Overhead

For a peripheral processor, Reflex introduces a small overhead in storage and memory, i.e., ~8KB ROM and ~0.2KB RAM, due to the runtime and the module library. Of all ROM and RAM equipped by one peripheral processor, the storage (ROM) overhead is less than 25% and the memory (RAM) overhead is less than 2.5%. The Reflex DSM further introduces little storage/memory overhead due to the compiler-inserted code and the object table. All `pre-access` or `post-access` code only increases the ROM usage of a peripheral module by at most 200 bytes. In each central module, the inserted code ranges from 200 to 700 bytes, less than 1.2% of the module storage usage. Each entry in the object table requires 6–8 bytes of extra RAM per module. The increased RAM is up to 119 bytes per module in our benchmarks.

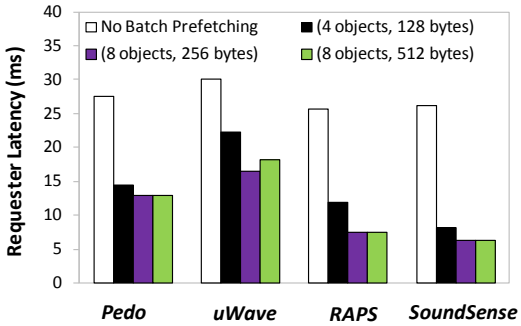


Figure 9. *The average requester latency in fetching an object, without and with batch prefetching*

8.4.3 DSM Execution Overhead

The DSM execution overhead comes from three major sources: module state checking as done by `pre-access`, requester latency, and home latency. In the common application scenarios where no inter-module data exchange occurs, the DSM execution overhead only comes from `pre-access`. With three ISAs in the tri-processor platform, each `pre-access` consists of one bit-shift operation, two to three load instructions and one compare instruction, resulting in only 5–10 extra processor cycles. Furthermore, the Reflex compiler aggressively eliminates `pre-access` for most accesses, using Mod/Ref analysis and object type information. We estimate that the state checking incurs negligible overhead (less than 3%) in the execution time of all benchmarks.

The DSM introduces home latency and requester latency. The measured home latency is 20ms on average, dominated by the message passing delay. Requester latency is effectively amortized over multiple objects by batch prefetching. To demonstrate its benefit, we compare the average requester latency per object without and with the batch prefetching. In the experiment, we vary batch parameters that specify the maximum size and the maximum count of objects that can be fetched with a single message. For example, batch parameters (4 objects, 128 bytes) specify that the DSM will flush a request message to the transport if objects requested by the message are more than 4 or their total size exceeds 128 bytes. As shown in Figure 9, the batch prefetching can reduce the average requester latency by up to 75% compared to the case without batch prefetching. In addition, we experimentally verify that without deferring *write_back*s, the DSM will slow down the execution of benchmark applications by up to 1000 times due to excessive messages, which is prohibitively expensive.

9. Related work

Reflex is the first work that seeks to enable developers to leverage low-power cores in mobile systems easily. The problem of programming heterogeneous, distributed systems is, however, not new. We next discuss the major approaches used to support heterogeneous system programming and in particular data exchange. It is important to keep in mind that Reflex faces a set of unique challenges when dealing with smartphones: *mature programming languages*, *energy efficiency* and *extreme architectural asymmetry*.

Programming with Message Passing: Most heterogeneous systems support inter-processor communication with message-like hardware primitives. Some directly expose message passing to application development. For example, Hydra [43] supports appli-

cation code distributed on multiple devices communicating with channels. Lime [11] supports code distributed on CPU and FPGA exchanging data through a buffer, a message-passing abstraction. Dandelion [15] supports applications that are distributed across a smartphone and wireless body-area sensors using RPC for message passing. As we analyzed in Section 2.3, programming with message passing can be tedious and error-prone for smartphone sensing applications.

Programming with DSM: Generally, the idea of creating a shared memory illusion by software with messages, or software DSM, is well-known. There is a rich body of software DSM designs for conventional loosely-coupled distributed systems like workstation clusters. Aiming at minimizing the performance overhead of shared memory, these DSM designs have employed OS kernel support [8], compiler techniques [33] and runtime systems [6, 14]. Also, heterogeneity has been addressed [44]. Most of them rely on hardware MMU support. While they inspire the design of the Reflex DSM, none of them is designed for energy efficiency or extreme architectural asymmetry. More recently, Hera-JVM [23] realizes a software cache among heterogeneous cores of the Cell processor. Unlike the Reflex DSM, it implements the cache coherence in a symmetric fashion and with a coarse granularity, by simply flushing the entire cache of a peripheral processor (i.e., SPE) to the main memory during synchronization. This makes Hera-JVM energy-inefficient and improper for smartphone sensing. Partitioned Global Address Space (PGAS) allows multiple processors (many homogeneous ones [2] or a few heterogeneous ones [31]) that share no coherent memory to logically share a window of shared addresses. Unlike Reflex, such a shared window is implemented with symmetric coherence protocols. ADSM [9] provides a software DSM between CPU and GPU by fully implementing the DSM on the CPU, a strategy opposite to the Reflex DSM. Although the strategy is asymmetric, it targets performance instead of energy efficiency and therefore will not work for smartphone sensing applications.

Custom Programming Model: Many programming models have been proposed for heterogeneous systems. Most of them define custom programming interface to bridge the developer's code with the underlying architecture. Qilin [20] provides an API for parallelizable computation so that the computation can be dynamically mapped to heterogeneous resources. Merge [17] defines new language constructs to support the map-reduce pattern. Cell-Ss [4] requires annotations of source code to exploit task-level parallelism. These custom programming models are successful in their targeted applications, mostly scientific or analytic workloads. In contrast, Reflex aims at supporting the mature style of smartphone application development, by providing an object-oriented class and requiring developers to encapsulate the code intended to be executed on peripheral processors.

Hardware-Coherent Memory: While most symmetric multiprocessing (SMP) systems support hardware-coherent memory as exemplified by commercial multicore processors, only a few heterogeneous systems [40, 41] rely on it. As we analyzed in Section 2, hardware-coherent memory is difficult, if possible at all, for the low-power peripheral processors that Reflex is concerned with.

10. Discussion

Type-safe Languages: The Reflex DSM leverages type information to aggressively reduce runtime overhead with static optimization. In the current prototype, we ensure that the type information is statically available by subsetting C++. A more ambitious implementation should directly support modern type-safe languages that are widely used in smartphone development, e.g., Java and C#. We notice that several existing systems already enabled type-safe languages for resource-constrained processors, e.g., Darjeeling [5], the .NET micro framework, and Hera-JVM [23]. We are working towards supporting type-safe languages under extreme resource scarcity.

Reflex beyond smartphone sensing: While Reflex and its DSM design are motivated by smartphones, their technical innovations have a broader impact. Many consumer electronic devices are embracing sensors, from ebook readers to game consoles to tablets. Reflex and its DSM design will allow developers to write efficient applications for such systems with ease. The core ideas of the Reflex DSM can be further extended to high-performance systems with loosely integrated, heterogeneous resources where energy efficiency has become an increasing practical concern.

11. Conclusion

Sensing applications are considered among the emerging killer applications for smartphones. Reflex is the first endeavor toward making programming heterogeneous, asymmetric smartphones easier. Sensing applications on heterogeneous smartphones pose unique systems challenges that were previously not important. In addressing these challenges, we have not only revised known solutions but also devised novel ones in software DSM that exploit extreme architectural asymmetry for high energy efficiency. We note that Reflex still imposes a few constraints in programming due to the requirement of encapsulating a peripheral module and the limit on the peripheral system services. However, our experience suggests such relaxation is necessary: eliminating these constraints would be too expensive to be practical, especially with such an asymmetric architecture.

Acknowledgments

This work was supported in part by NSF CAREER Award #1054693. Robert LiKamWa was supported by a Texas Instruments Graduate Fellowship. The authors are grateful for the useful comments made by the anonymous reviewers and the paper shepherd, Dr. Jan-Willem Maessen.

References

[1] Y. Agarwal, S. Hodges, R. Chandra *et al.*, "Somniloquy: augmenting network interfaces to reduce PC energy usage," in *Proc. USENIX Symp. Networked Systems Design & Implementation (NSDI)*, Boston, Massachusetts, 2009, pp. 365-380.

[2] C. Barton, C. Casaval, G. Almasi *et al.*, "Shared memory programming for large scale machines," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, Ottawa, Ontario, Canada, 2006, pp. 108-117.

[3] A. Baumann, P. Barham, P. Dagand *et al.*, "The Multikernel: A new OS architecture for scalable multicore systems," in *Proc. ACM Symp. Operating Systems Principles (SOSP)*, Big Sky, Montana, USA, 2009, pp. 29-44.

[4] P. Bellens, J. M. Perez, R. M. Badia *et al.*, "CellSs: a Programming Model for the Cell BE Architecture," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, Tampa, Florida, 2006, pp. 86.

[5] N. Brouwers, K. Langendoen, and P. Corke, "Darjeeling, a feature-rich VM for the resource poor," in *Proc. ACM Int. Conf. Embedded Networked Sensor Systems (SenSys)*, Berkeley, California, 2009, pp. 169-182.

[6] J. Carter, J. Bennett, and W. Zwaenepoel, "Implementation and performance of Munin," in *Proc. ACM Symp. Operating Systems Principles (SOSP)*, Pacific Grove, California, United States, 1991, pp. 152-164.

[7] D. Dhurjati, S. Kowshik, V. Adve *et al.*, "Memory safety without runtime checks or garbage collection," *ACM SIGPLAN Notices,* vol. 38, no. 7, pp. 69-80, 2003.

[8] B. Fleisch, and G. Popek, "Mirage: A coherent distributed shared memory design," *ACM SIGOPS Operating Systems Review,* vol. 23, no. 5, pp. 211-223, 1989.

[9] I. Gelado, J. E. Stone, J. Cabezas *et al.*, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *Proc. ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pittsburgh, Pennsylvania, USA, 2010, pp. 347-358.

[10] C. C. Han, M. Goraczko, J. Helander *et al.*, "CoMOS: An operating system for heterogeneous multi-processor sensor devices," *Technical Report MSR-TR-2006-117, Microsoft Research*, 2006.

[11] S. Huang, A. Hormati, D. Bacon *et al.*, "Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary," *ECOOP 2008 - Object-Oriented Programming*, Lecture Notes in Computer Science, pp. 76-103: Springer Berlin / Heidelberg, 2008.

[12] J. J. Labrosse, *MicroC OS II: The Real Time Kernel*: Newnes, 2002.

[13] C. Lattner, and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Int. Symp. Code Generation and Optimization (CGO)*, Palo Alto, California, 2004, pp. 75-86.

[14] K. Li, "Ivy: A shared virtual memory system for parallel computing," in *Proc. Int. Conf. Parallel Processing*, 1988, pp. 94-101.

[15] F. X. Lin, A. Rahmati, and L. Zhong, "Dandelion: a framework for transparently programming phone-centered wireless body sensor applications for health," in *ACM Wireless Health 2010*, San Diego, California, 2010, pp. 74-83.

[16] F. X. Lin, Z. Wang, R. LiKamWa *et al.*, "Transparent Programming of Heterogeneous Smartphones for Sensing," *Technical Report 0310-2011, Rice University*, 2011.

[17] M. D. Linderman, J. D. Collins, H. Wang *et al.*, "Merge: a programming model for heterogeneous multi-core systems," in *Proc. ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, USA, 2008, pp. 287-296.

[18] J. Liu, L. Zhong, J. Wickramasuriya *et al.*, "uWave: accelerometer-based personalized gesture recognition and its applications," *Pervasive and Mobile Computing,* vol. 5, no. 6, pp. 657-675, 2009.

[19] H. Lu, W. Pan, N. D. Lane *et al.*, "SoundSense: scalable sound sensing for people-centric applications on mobile phones," in *Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services (MobiSys)*, Kraków, Poland, 2009, pp. 165-178.

[20] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proc. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, New York, New York, 2009, pp. 45-55.

[21] D. Lymberopoulos, N. B. Priyantha, and F. Zhao, "mPlatform: a reconfigurable architecture and efficient data sharing mechanism for modular sensor nodes," in *Proc. ACM Int. Conf. Information Processing in Sensor Networks (IPSN)*, Cambridge, Massachusetts, USA, 2007, pp. 128-137.

[22] J. Manson, W. Pugh, and S. V. Adve, "The Java memory model," in *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL)*, Long Beach, California, USA, 2005, pp. 378-391.

[23] R. McIlroy, and J. Sventek, "Hera-JVM: a runtime system for heterogeneous multi-core architectures," in *Proc. ACM Int. Conf. Object Oriented Programming Systems Languages and Applications (OOPSLA)*, Reno/Tahoe, Nevada, USA, 2010, pp. 205-222.

[24] D. McIntire, K. Ho, B. Yip *et al.*, "The low power energy aware processing (LEAP) embedded networked sensor system," in *Proc. ACM Int. Conf. Information Processing in Sensor Networks (IPSN)*, Nashville, Tennessee, USA, 2006, pp. 499-457.

[25] E. B. Nightingale, O. Hodson, R. McIlroy *et al.*, "Helios: heterogeneous multiprocessing with satellite kernels," in *Proc. ACM Symp. Operating Systems Principles (SOSP)*, Big Sky, Montana, USA, 2009, pp. 221-234.

[26] NVIDIA, *NVIDIA Tegra 2*: http://www.nvidia.com/object/tegra-2.html.

[27] OKI Semiconductor, *ML8953A: MEMS 3-axis accelerometer*: http://www.okisemi.eu/Products/Sensors/mems_accelerometer.html.

[28] J. Paek, J. Kim, and R. Govindan, "Energy-efficient rate-adaptive GPS-based positioning for smartphones," in *Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services (MobiSys)*, San Francisco, California, USA, 2010, pp. 299-314.

[29] B. Priyantha, D. Lymberopoulos, and J. Liu, "LittleRock: Enabing Energy Efficient Continuous Sensing on Moble Phones," *Technical Report MSR-TR-2010-14, Microsoft Research*, 2010.

[30] Qualcomm, *Snapdragon MSM8660 and APQ8060 Product Brief*: http://www.qualcomm.com/documents/snapdragon-msm8x60-apq8060-product-brief.

[31] B. Saha, X. Zhou, H. Chen *et al.*, "Programming model for a heterogeneous x86 platform," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, 2009, pp. 431-440.

[32] Samsung, *Exynos 4210 Application Processor*: http://www.samsung.com/global/business/semiconductor/productInfo.do?fmly_id=844&partnum=Exynos%204210.

[33] D. Scales, K. Gharachorloo, and C. Thekkath, "Shasta: A low overhead, software-only approach for supporting fine-grain shared memory," *ACM SIGOPS Operating Systems Review,* vol. 30, no. 5, pp. 174-185, 1996.

[34] J. Sorber, N. Banerjee, M. D. Corner *et al.*, "Turducken: hierarchical power management for mobile devices," in *Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services (MobiSys)*, Seattle, Washington, 2005, pp. 261-274.

[35] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*: Morgan & Claypool, 2011.

[36] STMicroelectronics, *MEMS motion sensor 3-axis - ± 2g/± 8g smart digital output piccolo accelerometer*: http://www.st.com/stonline/products/literature/ds/12726/lis302dl.htm

[37] Texas Instruments, *OMAP5 Platform - OMAP5430*, http://www.ti.com/ww/en/omap/omap5/omap5-OMAP5430.html.

[38] Texas Instruments, "OMAP4 Applications Processor: Technical Reference Manual," 2010.

[39] Texas Instruments, "TI SYS/BIOS Real-time Operating System v6.x User's Guide (Rev. I)," 2010.

[40] G. Venkatesh, J. Sampson, N. Goulding *et al.*, "Conservation cores: reducing the energy of mature computations," in *Proc. ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pittsburgh, Pennsylvania, USA, 2010, pp. 205-218.

[41] P. H. Wang, J. D. Collins, G. N. Chinya *et al.*, "EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, San Diego, California, USA, 2007, pp. 156-166.

[42] R. Want, "Invited Talk: Always-on Considerations for Mobile Systems," in *DAC Wrkshp. Mobile and Cloud Computing*, 2010.

[43] Y. Weinsberg, D. Dolev, T. Anker *et al.*, "Tapping into the fountain of CPUs: on operating system support for programmable devices," in *Proc. ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, USA, 2008, pp. 179-188.

[44] S. Zhou, M. Stumm, K. Li *et al.*, "Heterogeneous distributed shared memory," *IEEE Trans. Parallel and Distributed Systems,* vol. 3, no. 5, pp. 540-554, 2002.