Heterogeneous Distributed Shared Memory for Lightweight Internet of Things Devices

IoT-HDSM is a compiler-runtime cooperative heterogeneous distributed shared memory (HDSM) framework in which the compiler unifies heterogeneous memory layouts of different IoT devices and the runtime system provides a shared memory view of the devices. This article describes an implementation of three IoT services with IoT-HDSM and shows that IoT-HDSM simplifies shared data management without harming quality of service.

Bongjun Kim
Seonyeong Heo
Gyeongmin Lee
Soyeon Park
Hanjun Kim
Jong Kim
Pohang University of Science and Technology

Despite the promising applicability of the Internet of Things (IoT), it is challenging for programmers to build an IoT application because of its heterogeneous and distributed execution environments. In an IoT application, diverse IoT devices collaboratively collect and share data to provide enriched services. For example, a fitness-tracking application in Figure 1 consists of a scale, a smartphone, and a server. It measures different tracking features such as weights, body mass index (BMI), step counts, and walking distance and suggests personalized fitness and diet plans. To integrate the diverse IoT devices into one service, programmers should deeply understand the service application, analyze shared data among the devices, and explicitly insert communication codes for the shared data. Moreover, since underlying architectures of the IoT devices are heterogeneous, programmers should also insert translation codes to integrate heterogeneous memory layouts.

To reduce the burden on IoT programmers, recent proposals such as network embedded systems C (nesC),[1] Node-RED (http://nodered.org), OpenRemote (www.openremote.org), Smart Home (http://eclipse.org/smarthome), and the Thing System (http://thethingsystem.com) provide communication APIs and graphical tools that simplify communication management (see also the "Related Work in Integrating Heterogeneous Devices" sidebar). However, the APIs and graphical tools cannot fully liberate programmers from communication management, because programmers still must specify explicitly how to communicate shared variables among IoT devices. Node-RED provides a shared memory space for global variables to programmers, but its programming model is limited to the unidirectional dataflow approach.

Addressing the challenges in IoT programming, this article revisits heterogeneous distributed shared memory (HDSM) for IoT applications and proposes a compiler-

Published by the IEEE Computer Society

Related Work in Integrating Heterogeneous Devices

Node-RED (http://nodered.org) and its extensions, such as Distributed Node-RED,[1] and glue.things (www.gluethings.com), are visual IoT programming tools that provide a holistic view of a service and facilitate communication among devices. With the built-in library, programmers specify dataflows by creating nodes and wiring them for communication. Moreover, Node-RED lets all nodes share global variables through the global context. However, Node-RED's programming model is limited to the dataflow approach. On the other hand, IoT-HDSM does not compel any logical program structure, which makes it relatively more flexible on data sharing.

Heterogeneous distributed shared memory (HDSM) is an extended DSM system for heterogeneous system environments. To overcome the heterogeneity of machine architectures and programming languages, HDSM applies data conversion when data is transferred between hosts at runtime. Songnian Zhou and colleagues show that HDSM can maintain the functional transparency of homogeneous DSM.[2] Despite its extensibility, HDSM is unsuitable for an IoT environment due to its runtime overhead, which is relatively high for lightweight devices. Unlike HDSM, IoT-HDSM unifies heterogeneous memory layouts at compile time and reduces the translation overheads.

Reflex is a compiler and runtime framework to ease programming for heterogeneous hardware architecture in a smartphone.[3] Its core is its own software DSM design to leverage the architectural asymmetry and minimize performance overhead. Reflex allows data exchange between heterogeneous processors through procedure call or shared memory on the Reflex DSM. This approach to memory sharing is similar to IoT-HDSM, but Reflex focuses only on interprocessor communication in a device, whereas IoT-HDSM targets communication between heterogeneous devices.

PiMiCo proposes a new compiler-assisted programming model that reduces data communication costs and preserves privacy in mobile-cloud systems.[4,5] Because IoT applications manage different private data and IoT devices are lightweight, efficient algorithms for preserving privacy are crucial for IoT platforms. PiMiCo complements IoT-HDSM to support additional features, such as privacy protection.

References

1. N.K. Giang et al., "Developing IoT Applications in the Fog: A Distributed Dataflow Approach," *Proc. 5th Int'l Conf. Internet of Things*, 2015; doi:10.1109/IOT.2015.7356560.
2. S. Zhou et al., "Heterogeneous Distributed Shared Memory," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, no. 5, 1991, pp. 540–554.
3. F.X. Lin et al., "Reflex: Using Low-Power Processors in Smartphones Without Knowing Them," *Proc. 17th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 13–24.
4. K. Ravichandran, A. Gavrilovska, and S. Pande, "PiMiCo: Privacy Preservation via Migration in Collaborative Mobile Clouds," *Proc. 48th Hawaii Int'l Conf. System Sciences*, 2015, pp. 5341–5351.
5. K. Zhang and S. Pande, "Efficient Application Migration under Compiler Guidance," *Proc. ACM SIGPLAN/SIGBED Conf. Languages, Compilers, and Tools for Embedded Systems*, 2005, pp. 10–20.

runtime cooperative lightweight HDSM framework called IoT-HDSM. The compiler inserts translation codes for heterogeneous memory structures, address sizes, and endianness of different IoT devices, and the runtime system provides a shared memory view across the devices. For lightweight IoT devices that cannot execute the runtime, the compiler additionally inserts explicit communication codes for shared data.

To reduce memory consistency overheads of IoT-HDSM, this work also proposes a new concurrency annotation called a *memory branch*. In an IoT application, some shared data are correlated and should be updated together at the same time. For example, the fitness-tracking application in Figure 1 should read a step count and its walking distance together at the same time to correctly calculate calories burned. The proposed memory branch provides a memory snapshot for an IoT device at the beginning of the branch and merges all the updated values into the master memory at the end of the branch. If two devices update the same variables in a branch with different values at the same time, IoT-HDSM reflects the values in the last branch into the master memory. Although the memory branch does not guarantee mutually exclusive accesses on the shared variables, unlike locks, the branch lets each IoT device concurrently access the shared variables on a memory snapshot without suffering from blocking.

This article implements the IoT-HDSM framework on top of LLVM C/C++

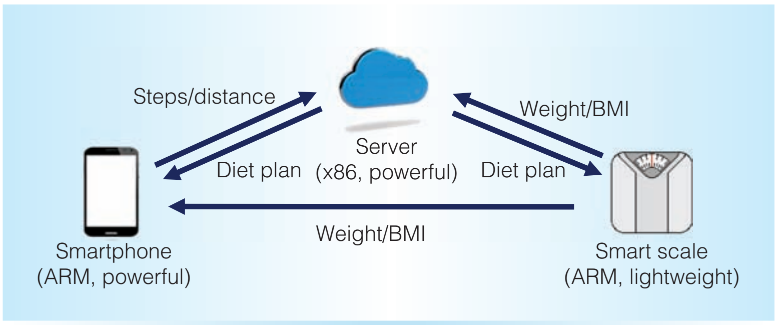

Figure 1. Fitness-tracking application. A server collects personal information such as weights, body mass index (BMI), step counts, and walking distance from a scale and a smartphone, and suggests personalized fitness and diet plans to a user.

compiler infrastructure and uses three IoT services—fitness tracking, calling a taxi, and heart attack detection—with 10 events to evaluate IoT-HDSM using real IoT devices. The evaluation results show that the IoT-HDSM framework liberates programmers from communication management without affecting the quality of service (QoS) of IoT services.

IOT-HDSM Programming Model

In this section, we use a simple code example to show how the shared memory view can simplify IoT application development, and we explain the necessity and the operation model of the proposed memory branch annotation.

Shared Memory View of IoT-HDSM

Without a distributed shared memory system, developing an IoT application requires huge amounts of programmer effort due to explicit communication management for shared data. Because multiple devices collaboratively execute a service function in an IoT application, each device could use data that other devices generate. To implement the IoT application, programmers should know how the data is generated and used among the devices and should explicitly insert communication codes between the devices to share the data. For example, to develop a fitness-tracking application, the programmers should know that a scale and a smartphone collect weight, BMI, step count, and walking distance data, and that a server needs the collected data to make a diet plan. From this knowledge, the programmers design a communication model (see Figure 1) and insert communication codes such as `send` and `recv` in Figure 2a among the IoT devices. If generated data are used in multiple devices, programmers need to insert the communication codes multiple times. For example, `scale` invokes the send function multiple times to deliver `weight` and `bmi` to `smartPhone` and `server` in Figure 2a. Although recent proposals have simplified communication management with communication APIs and graphical tools, IoT programmers still must explicitly manage the communication.

IoT-HDSM provides a shared memory view for distributed IoT devices and liberates IoT programmers from the explicit communication management of shared data. IoT-HDSM makes global and heap allocated variables automatically shared across IoT devices while keeping stack variables at local devices. Therefore, by allocating shared variables at global or heap memory and local variables at stack memory, programmers can easily manipulate which variable will be shared in the program, and can develop the application without worrying about communication codes. Figure 2b shows how much IoT-HDSM simplifies the fitness-tracking program. Programmers can make `weight` and `bmi` shared among `scale`, `smartPhone`, and `server` by declaring `user` as a global variable and storing the value at heap without any `send` and `recv` function call. Here, if an IoT device needs to access a variable that is defined by another device, the programmers define and use the variable with the `extern` keyword, such as `line 2` of the `scale` program.

Memory Branch

As IoT-HDSM lets multiple devices concurrently access a shared variable, concurrency control becomes crucial for the correct execution. Unlike the explicit message-passing programming model, in which programmers can control when to send locally updated values to other devices, IoT-HDSM lets other devices access a shared value in the middle of updates, and causes a correctness problem. For example, in the message-passing program, `scale` transfers updated `weight` and

```
\* *** Data Types *** *\
typedef struct {
 AccountInfo *account;
 float weight; float bmi;
 int steps; float distance;
 Plan *dietPlan;
} User;

\* *** Smart Scale *** *\


void scaling() {

 float weight = measureWeight();
 float bmi = measureBMI();

 send(server, weight, bmi);
 send(smartPhone, weight, bmi);
}

void recvDataFromServer() {
 recv(server, &user->dietPlan);
}

\* *** Server *** *\
User *user;

void analyze() {

 user->dietPlan = makeDietPlan(user->weight,
          user->bmi, user->steps,
          user->distance);

 send(smartPhone, user->dietPlan);
 send(scale, user->dietPlan);
}

void recvDataFromScale() {
  recv(scale, &user->weight, &user->bmi);
}

void recvDataFromSmartphone() {
  recv(smartPhone, &user->steps, &user->distance);
}
```

(a)

```
\* *** Data Types *** *\
typedef struct {
  AccountInfo *account;
  float weight; float bmi;
  int steps; float distance;
  Plan *dietPlan;
} User;

\* *** Smart Scale *** *\
extern User *user;

void scaling() {
  @branch {
    user->weight = measureWeight();
    user->bmi = measureBMI();
  }
~~  send(server, weight, bmi);~~
~~  send(smartPhone, weight, bmi);~~
}

~~void recvDataFromServer() {~~
~~  recv(server, &user ->dietPlan);~~
~~}~~

\* *** Server *** *\
User *user;

void analyze() {
  @branch {
    user->dietPlan = makeDietPlan(user->weight,
    user->bmi, user->steps,
    user->distance);
  }
~~  send(smartPhone, user ->dietPlan);~~
~~  send(scale, user ->dietPlan);~~
}

~~void recvDataFromScale() {~~
~~  recv(scale, &user ->weight, &user ->bmi);~~
~~}~~

~~void recvDataFromSmartphone() {~~
~~  recv(smartPhone, &user ->steps, &user ->distance);~~
~~}~~
```

(b)

Figure 2. Code example: simplified fitness tracking (parts of the smart scale and server codes only). (a) Message-passing program. (b) Program on IoT-HDSM. IoT-HDSM liberates programmers from explicit communication management by providing a shared memory view.

`bmi` to `server` at the same time (see the gray boxes in Figure 3a), so `server` always accesses `weight` and `bmi` in the same version and calculates a correct diet plan. However, in the IoT-HDSM program without any concurrency control, `server` can access `weight` and `bmi` in different versions (the gray and white boxes in Figure 3a), so `server` may generate an incorrect diet plan (the slashed boxes in Figure 3b).

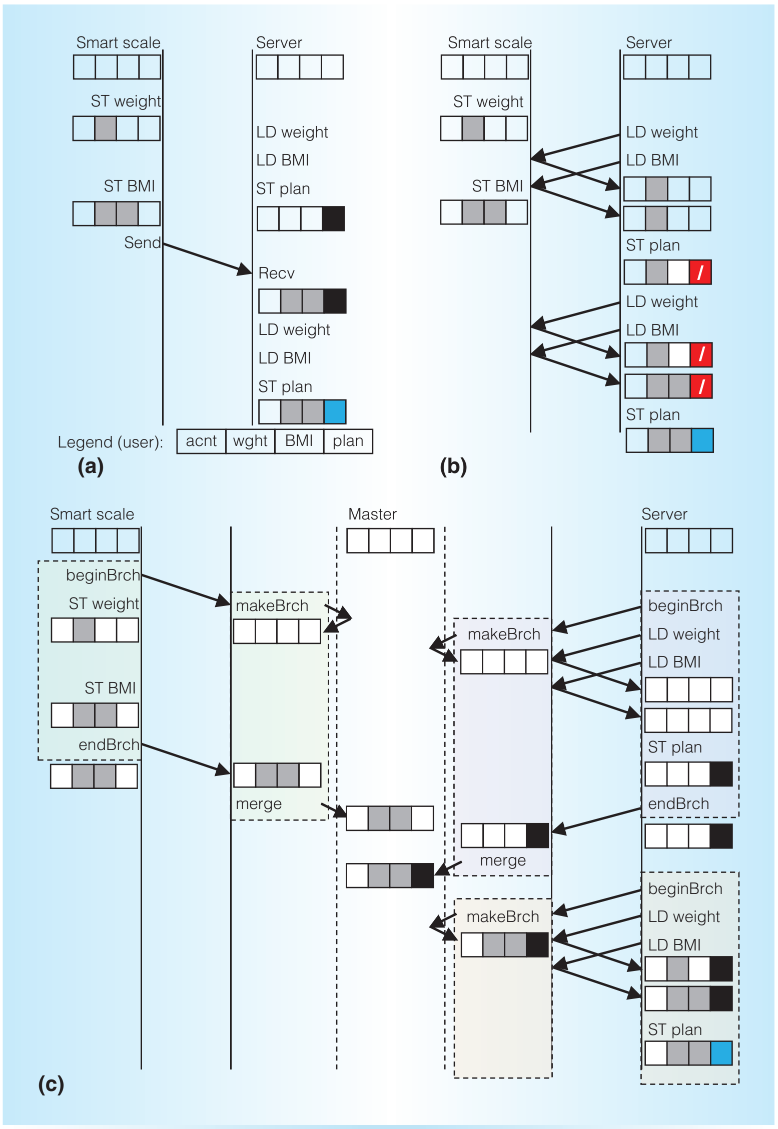

Figure 3. Timing graphs of communicating shared data. (a) Message passing; (b) IoT-HDSM without branch; (c) IoT-HDSM with branch. Without the memory branch annotation, the server could read a partly updated set of data, such as gray weight and white BMI variables. The branch annotation allows the smartphone and the server to operate on a snapshot of a shared memory.

Although locks and atomic regions can solve the concurrency problem by atomically updating a group of shared variables, they could cause QoS problems. Because many IoT devices are lightweight, supporting only a single thread, a lock could block all the operations, including sensing data in the devices, and the devices could fail to provide a service or miss some important sensing data. For example, if `server` has a lock on the `user` variable, `scale` cannot measure the user's weight, and `smartPhone` cannot update `steps` until `server` releases the lock. Transactional memory can allow multiple devices to concurrently access shared data, but because transactional memory requires huge memory spaces for validation and rollback, it is not suitable for lightweight IoT devices.

To support lightweight update of a group of shared data, this article introduces a new annotation, called a *branch*. As with repository branches in version control systems such as Subversion and Git, IoT-HDSM takes a memory snapshot when an IoT device enters a branch region. In the branch region, the device executes the program only on the snapshot without any coherence operation. When the branch region ends, IoT-HDSM atomically merges updated values to globally shared memory. IoT-HDSM keeps a master node to maintain the up-to-date globally shared memory and branch versions. Figure 3c illustrates how the branch region works for the fitness-tracking example. When `scale` enters a branch region, `scale` requests a branch snapshot to the master and updates `weight` and `bmi`. Because the updates are in the branch region, `server` cannot see the updated values, and it reads `weight` and `bmi` in the same version (white boxes). When `scale` exits the branch region, IoT-HDSM merges the updated `weight` and `bmi` to the master. After the merge, `server` can see the updated `weight` and `bmi` (gray boxes).

The branch annotation is valid only for updating correlated shared variables without serializability. Although many IoT applications are required to update correlated shared variables together at the same time, the applications do not care about the serializability of the updates. For example, while the branch annotation updates a set of variables together in the annotated region and prevents the applications from generating an incorrect result—such as the slashed boxes in Figure 3b—the annotation does not abort a result from conflicted branches, such as the black boxes in Figure 3c. Although the weight and BMI are already updated as the gray boxes in the master node, the master accepts the diet plan (the black box) that is based on the old values. The branch annotation can be correctly used here because reflecting which set of data is used is not important for generating the correct diet plan if a user measures his weights and BMIs multiple times (with multiple scales). If two devices concurrently update the same shared variable and its serializability is crucial, programmers should use a lock or atomic region instead of the branch.

IOT-HDSM Framework

Figure 4 illustrates the overall structure of the IoT-HDSM framework for providing heterogeneous distributed shared memory to IoT devices. First, the front-end compiler parses annotations about the branch regions and computing power of IoT devices. Second, the compiler unifies heterogeneous memory layouts of various IoT devices. Because the architectures of the IoT devices in an application are heterogeneous, their memory layouts, such as address sizes and structure alignments, are also heterogeneous. The compiler statically unifies the memory layout and minimizes the translation overheads at runtime. Third, the compiler reallocates global and heap variables into shared memory spaces. Here, the compiler inserts explicit communication codes for lightweight devices that cannot support the IoT-HDSM runtime. Finally, the IoT-HDSM runtime supports memory coherence operations across distributed IoT devices.

Memory Unification

Although distributed IoT devices have the same shared memory space through IoT-HDSM, the devices cannot correctly share the same value because of heterogeneous memory layouts. For example, in the fitness-tracking application in Figure 2, a server and a scale allocate the same object `user` with different memory layouts, because the server is a 64-bit machine and requires 8 bytes for its pointer variables, whereas the scale is a 32-bit device and requires 4 bytes for pointers. As a result, the server and the scale can use different memory addresses for the same variable, requiring expensive address translation overheads.

To reduce the translation overheads at runtime, IoT-HDSM unifies the heterogeneous memory layouts of IoT devices at compile time. The IoT-HDSM compiler realigns the heterogeneous memory structures as a unified structure. If there is a pointer with different sizes, such as a 32-bit pointer and a 64-bit pointer in a structure, the compiler extends the 32-bit pointer type to the 64-bit pointer type. Although ARM and x86 can have the same endianness, the compiler inserts endianness translation codes for each memory access if an IoT device uses a different endianness. The memory layout unification is similar to previous work[2] except for the memory pointer unification; the

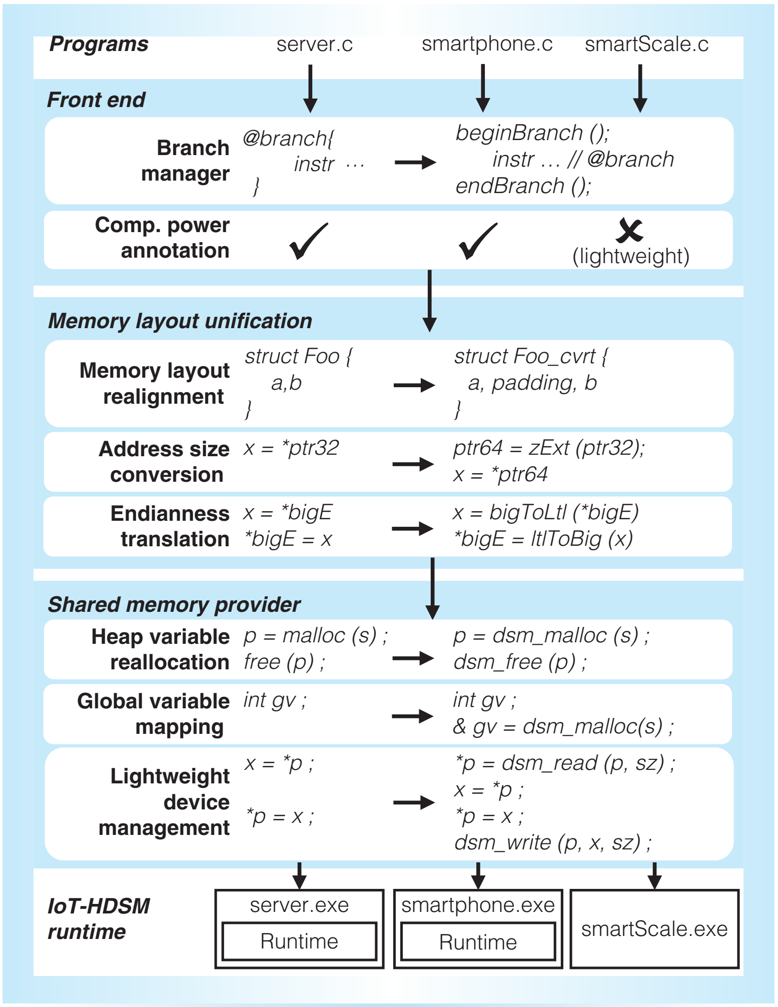

Figure 4. Overall structure of IoT-HDSM framework with the programs in Figure 2. The IoT-HDSM compiler modifies data structures and inserts translation instructions to integrate heterogeneous memory layouts. The compiler inserts explicit communication codes for shared data accesses to support lightweight devices that cannot execute the IoT-HDSM runtime.

Table 1. Benchmark specification.

Benchmark	Device	Specification
`Fitness Tracker`	Server	Desktop server (Intel Core i7-6700)
	Smartphone	Samsung Galaxy S5 (Qualcomm Snapdragon 801)
	Smart scale	ODROID-XU4 (Samsung Exynos 5422)
`Taxi App`	Server	Desktop server
	Taxi	ODROID-XU4
	Customer	Samsung Galaxy S5
`Heart Attack`	Smartphone	ODROID-XU4
	911 server	Desktop server
	Hospital	Desktop server

IoT-HDSM framework newly supports memory versioning through the memory branch annotation. Moreover, to effectively support lightweight IoT devices with 8/16-bit microcontrollers, the compiler transforms 64-bit pointer variables to 8- or 16-bit dereferenced variables if possible and inserts explicit communication codes for the dereferenced variables.

Shared Memory Management

The IoT-HDSM framework provides shared memory spaces for global and heap variables. For heap variables, the IoT-HDSM compiler replaces existing allocation and deallocation call sites with IoT-HDSM allocation and deallocation function calls. For global variables, the IoT-HDSM compiler reallocates the global variable area to the DSM space and initializes the addresses of the global variables as the DSM addresses for all the IoT devices. Moreover, to minimize the coherence overheads, IoT-HDSM adopts the lazy release consistency model[3,4] as a underlying coherence protocol.

Some IoT devices are not powerful enough to execute the IoT-HDSM runtime. For example, because lightweight IoT devices cannot support multithreading, IoT-HDSM cannot execute its runtime daemon as a background process. Although an IoT device can support multithreading, it can be more efficient to explicitly communicate shared data than to rely on the shared memory runtime, especially if the device accesses only a few data, as with the smart scale in Figure 1. For lightweight devices, the IoT-HDSM compiler automatically inserts explicit communication codes for every shared memory access, so the devices can exploit the shared memory view without the runtime.

Memory Branch Management

Because the IoT-HDSM compiler unifies heterogeneous memory layouts, the IoT-HDSM runtime provides only coherence protocols for shared objects across distributed IoT devices, such as existing software DSM.[3–6] Additionally, to support the memory branch annotation, the runtime keeps up-to-date memory states at the master node. The most powerful computing node becomes the master node, as with the cloud server in Figure 1.

For every memory branch annotation, the front-end IoT-HDSM compiler inserts `beginBranch` and `endBranch` function calls at the beginning and end of the branch region. When a device invokes `beginBranch`, the device sends a branch entrance signal to the master node, and the master takes a snapshot of the current memory state by creating a new process that has a cloned memory space but separate memory. Whenever the device requests memory values, the master node sends the value from the process memory. The device keeps a write set for updated values in the memory branch region and sends the write set with a branch merge signal at the end of the memory branch. The master directly and atomically updates write sets at the master memory and kills the branch process. If there is no load instruction in a branch region, as in the scale function in

Figure 2, the compiler marks the branch region as a write-only branch and does not generate a process because the memory snapshot is not necessary.

Evaluation

To evaluate IoT-HDSM, we implemented three C++ IoT applications—a fitness tracker, a taxi service, and heart-attack detection—with 10 events in message passing and shared memory programming models (with and without branch annotation). As Table 1 shows, each service uses various IoT devices, such as an embedded system board (ODROID-XU4) with Samsung Exynos 5422, a state-of-the-art Samsung Galaxy S5 smartphone, and a desktop server with Intel Core i7-6700. This work assumes that the embedded board is lightweight, whereas the smartphone and desktop are powerful.

Without any correctness or QoS problem, the IoT-HDSM framework successfully unifies heterogeneous memory layouts across IoT devices and liberates programmers from shared data management. Figure 5 shows each request's response time in IoT services. Although IoT-HDSM suffers from a geomean slowdown of 2.98 times compared to message passing due to round-trip communication for memory coherence, the average and maximum response time overheads are 24 and 100 ms, respectively, so IoT-HDSM does not harm the IoT applications' QoS. Figure 5 also shows that the memory branch annotation reduces the coherence overheads and achieves the geomean speedup of 1.33 times.

Figure 6 shows how IoT-HDSM simplifies IoT programming with lines of code comparison. Compared to message-passing programming models, the shared-memory programming model of IoT-HDSM requires 35.36 percent fewer lines of code. The difference comes from the automatic communication management and memory layout translation among heterogeneous IoT devices. By taking responsibility for complex communication management and memory layout translation, the IoT-HDSM framework makes IoT programming easy.

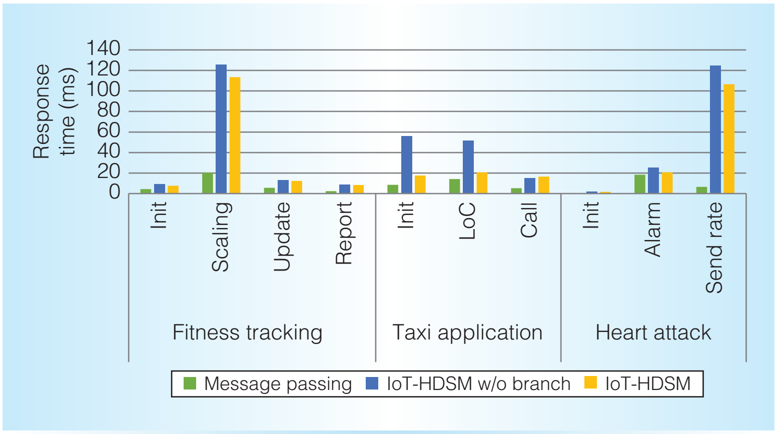

Figure 5. Response time for each event in IoT applications. Compared to the message passing model, the IoT-HDSM framework suffers from 24 ms and 100 ms average and maximum latency overheads. The memory branch annotation yields the geomean speedup of 1.33 times beside the non-branch version.

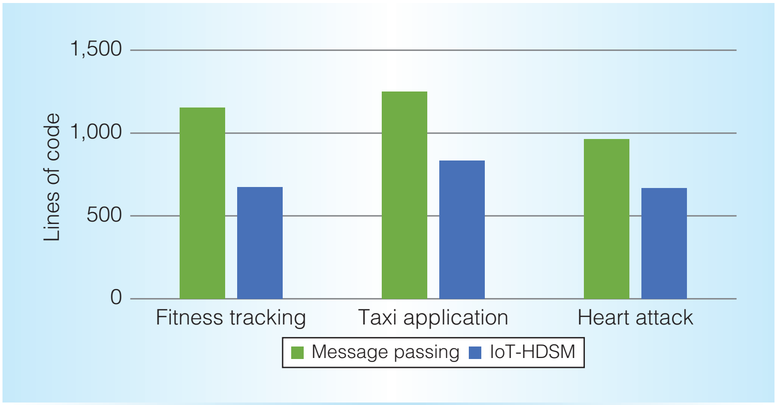

Figure 6. Lines of code (LoC) of IoT applications. IoT-HDSM simplifies the IoT programming by requiring 35.56 percent fewer lines of code than the message passing model.

Although the IoT-HDSM framework efficiently simplifies communication management by integrating heterogeneous and distributed memory spaces of IoT devices, the seamless memory integration increases the risk of information leakage. Privacy protection is crucial for IoT services because many IoT services collect personal information such as health and medical information. However, IoT sensors and actuators are too lightweight to support existing heavy privacy protection algorithms, and many IoT service programmers are not security experts. Therefore, a lightweight and easy information protection mechanism is necessary for IoT programmers and platforms, including IoT-HDSM. This work leaves the security challenge as a future work. MICRO

Acknowledgments

We thank the anonymous reviewers and editors for their insightful comments and suggestions. We also thank CoreLab and HPC Lab for their support and feedback during this work. This work is supported by Samsung Research Funding Center of Samsung Electronics under project no. SRFC-TB1403-04.

References

1. D. Gay et al., "The nesC Language: A Holistic Approach to Networked Embedded Systems," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2003; doi:10.1145/781131.781133.
2. G. Lee et al., "Architecture-Aware Automatic Computation Offload for Native Applications," *Proc. 48th IEEE/ACM Int'l Symp. Microarchitecture*, 2015, pp. 521–532.
3. P. Keleher, A.L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, 1992, pp. 13–21.
4. Y. Zhou, L. Iftode, and K. Li, "Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems," *Proc. 2nd USENIX Symp. Operating Systems Design and Implementation*, 1996, pp. 75–88.
5. J.B. Carter, "Design of the Munin Distributed Shared Memory System," *J. Parallel and Distributed Computing*, Sept. 1995, pp. 219–227.
6. D.J. Scales and K. Gharachorloo, "Towards Transparent and Efficient Software Distributed Shared Memory," *Proc. 16th ACM Symp. Operating Systems Principles*, 1997, pp. 157–169.

Bongjun Kim is a PhD student in the Department of Computer Science and Engineering at Pohang University of Science and Technology (POSTECH). His research focuses on compiler optimization for the Internet of Things and security. Kim received a BS in computer science and engineering from POSTECH. Contact him at bong90@postech.ac.kr.

Seonyeong Heo is a PhD student in the Department of Computer Science and Engineering at Pohang University of Science and Technology (POSTECH). Her research interests include compiler optimization for dynamic software updating and the Internet of Things. Heo received a BS in computer science and engineering from POSTECH. Contact her at heosy@postech.ac.kr.

Gyeongmin Lee is a PhD student in the Department of Creative IT Engineering at Pohang University of Science and Technology (POSTECH). His research focuses on the programmability of IoT applications. Lee received a BS in computer science and engineering from POSTECH. Contact him at paina@postech.ac.kr.

Soyeon Park is an undergraduate student in the Department of Computer Science and Engineering at Pohang University of Science and Technology (POSTECH). Her research interests include computer systems and security. Contact her at thdusdl1219@postech.ac.kr.

Hanjun Kim is an assistant professor in the Departments of Creative IT Engineering and Computer Science and Engineering at Pohang University of Science and Technology (POSTECH). His research interests include computer architecture and compiler optimization for distributed and emerging systems. Kim has a PhD in computer science from Princeton University. Contact him at hanjun@postech.ac.kr.

Jong Kim is a professor in the Department of Computer Science and Engineering at Pohang University of Science and Technology (POSTECH). He is also a director of POSTECH Information Research Laboratories. His research interests include system and network security, mobile embedded software, dependable computing, and parallel and distributed computing. Kim received a PhD in computer engineering from Pennsylvania State University. Contact him at jkim@postech.ac.kr.