Abstract
Transparent Computing (TC) can be thought as a special kind of cloud computing that regards storage as a service. TC logically splits the software stack from the underlying hardware platform, and separates the computing unit from storage for the purpose of making the same software run on different hardware and different software run on the same hardware. TC requires a unified software-hardware interface to abstract the underlying platform details. As the new generation BIOS interface, Unified Extensible Firmware Interface (UEFI) defines the interface between the software stack and hardware platform, which aligns TC well. UEFI’s modular design also allows for BIOS customization. An ideal vision of TC implementation with UEFI is proposed in this paper as the forecast. Several models are also provided to illustrate the constraints of current hardware and software stack in terms of performance, portability, and hardware adaptability. Finally, a case study of a wireless TC tablet is provided together with the analysis.

1. Introduction
With the proliferation of high-speed computer networks, high-performance computer systems, and high-capacity storage devices, the concept of cloud computing has become quite popular. Cloud computing can be viewed as the successor and modern implementation of the outdated mainframe system, where the dumb terminal is replaced by a computing device at client side, the serial cable connection between terminal and mainframe is extended through a high-speed computer network, the low performance central processing unit is upgraded to a powerful distributed “computer cloud,” and the hard disk is replaced with huge storage devices. Cloud computing is the delivery of computing as a service rather than a product, whereby shared resources, software, and information are provided to computers and other devices as a utility (like the electricity grid) over a network (typically the Internet) (wikimedia, online). Similar to the antiquated mainframe, cloud computing follows the client-server architecture. Figure 1 shows the concept diagram and classifications of cloud computing. According to the different services provided by the server (infrastructure, platform, and application), different types of service architecture (Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS)) come into being.

Cloud computing provides a means of resource sharing and service provisioning via network. However, how to re-use current software resources designed for standalone computers has become an issue facing cloud computing deployment. Most cloud computing implementations are Software as a Service (SaaS) with web-based software architecture—they run the service software at the remote server and redirect results to the client via web browser. However, current standalone software systems are seldom designed for web architecture,
and converting all standalone software to make them web-based is almost impossible. This condition leads to the question of how we can reuse standalone software from the cloud. One possible solution is to treat the storage as a service from the network, and logically access a remote virtual disk as if it was physically attached to the client. This concept reaches a new computing paradigm: Transparent Computing (TC).

TC can be thought as a special kind of cloud computing that provides software storage as a service, or to be more specific, a special kind of IaaS that treats permanent storage as a kind of infrastructure. TC logically splits the software stack from the underlying hardware platform, and separates the computing unit from the permanent storage for the purpose of making the same software run on different hardware and different software run on the same hardware (Chen and Zheng, 2008). Thus, finding a suitable logical interface between hardware and software will be the new challenge in TC implementation.

As the next-generation firmware interface, Unified Extensible Firmware Interface (UEFI) defines the interface between operating system and platform firmware (Unified EFI Inc., 2011). UEFI aligns TC with the system architecture, and provides a unified interface with which to abstract the underlying hardware and separate the software stack from the hardware platform. UEFI has already been adopted by several industries, and is a popular standard interface. In addition, UEFI’s modular design provides a means of customizing BIOS by adding value-add modules to the firmware.

Although UEFI appears to be a good choice for TC implementation, a few issues exist regarding how TC works with current hardware and software stack: UEFI works only during the pre-boot period and not at run-time, and the OS does not call UEFI service, but accesses the hardware register directly. Solutions need to be found for these limitations. Furthermore, to embrace trends such as wireless and embedded devices like tablets and mobile phones, we need to consider how TC can be made to work well with such technology in a wireless environment. New challenges like network reliability, device loss, data protection, and application/OS changes ought to be carefully considered.

In this paper, the TC concept is reviewed in Chapter 2, and a brief UEFI introduction is given in Chapter 3. The paper then describes the ideal implementation of a UEFI-based TC solution in Chapter 4, and several trade-off solutions considering the constraints of available hardware and software will be discussed with the appropriate analysis in Chapter 5. In Chapter 6, a real-world case study based on WiFi and tablet is discussed and analyzed. Chapter 7 summarizes and gives the study’s conclusion regarding the value of UEFI to TC implementation. Finally, the acknowledgements are provided.

2. Brief Review of the Transparent Computing Concept

As previously mentioned, TC can be thought as a special kind of cloud computing where storage is treated as a service. The major characteristic of Transparent Computing involves two “separations”: the separation of software stack and hardware platform, and the separation of computing and storage. The purpose of TC consists of two “runs”: to run the same software on different hardware platforms, and to run different software on the same hardware platform. Just like cloud computing, TC also follows the client-server architecture. TC is the implementation of pervasive or ubiquitous computing. The execution of computer instruction and data is temporally and spatially separated from their storage (Zhou and Zhang, 2010; Zhang, 2004).
The main components of Transparent Computing are shown in Figure 2:

- TC client: the client part of the TC infrastructure, which in most cases is bare-metal for running different software stacks on the top;
- TC server: the backend of TC infrastructure. TC servers are mostly used for the storage and provision of software services to clients. Other value-add features like infrastructure management, user identification, and security are also provided here;
- TC delivery network: the network connection between TC client and TC server. The bandwidth of the TC delivery network will significantly impact the overall performance.

As mentioned above, TC shares some common characteristics with cloud computing, including separation of computing and storage, client-server architecture, implementation of SaaS, dynamic management of software resource, transparent software maintenance, and less management effort required for a group of clients. However, in Cloud Computing, the data flow is from server to client—computing and storage is done at the server and displayed locally. In TC, however, data go from client to server—data are generated by local computing and then put in remote storage. The TC client is bare-metal and the software, as a service, is piped from remote TC servers via the high-speed network, as in our water supply or power grid. Stream execution of the OS and upper software is another characteristic of TC (Zhang and Zhou, 2010).

<table>
<thead>
<tr>
<th></th>
<th>Cloud Computing</th>
<th>Transparent Computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computing at</td>
<td>Remote</td>
<td>Local</td>
</tr>
<tr>
<td>Storage at</td>
<td>Remote</td>
<td>Remote</td>
</tr>
<tr>
<td>Display at</td>
<td>Local</td>
<td>Local</td>
</tr>
</tbody>
</table>

Table 1 summarizes the key differences between Cloud Computing and TC. In brief, Transparent Computing only utilizes the remote storage, but in Cloud Computing, everything is finished at the remote server except for the display. Transparent Computing utilizes the local hardware resources more than general Cloud Computing.

Compared with cloud computing, TC brings special advantages: client side computing that makes full use of hardware, stream execution that does not change the user experience, compatibility with existing software environments, and increased system security due to remote storage.

With TC, end-users do not need to care about software maintenance, and can select different computing services as simply as selecting different television channels. The reason is that the software environments provided by TC servers can be freely chosen given its bare-metal approach. This flexibility achieves the concept of SaaS and gives new meaning to “transparency” in Transparent Computing.

From the hardware viewpoint, the obvious benefit of TC is the increased efficiency in both computing and storage. The bare-metal does not need to carry as much software as traditional standalone PCs, but only needs to fetch the necessary software for local use. This aspect significantly reduces the efforts for software loading and scheduling, thus improving overall efficiency. This process, in turn, leads to the possibility of running large software on small hardware, for example, running resource-intensive games on entry-level mobile phones.

Regarding the enhancement of computer architecture, TC extends the novel Von-Neumann architecture that has dominated the field for decades by sending permanent storage to remote—either in logical partitions of a huge storage server, or in virtual partitions composed of free disk space from thousands of computers. Most importantly, the “bus” here is the high-speed computer network to which the central process unit is connected. TC infrastructure can be viewed as a huge PC dynamically bound by different peripherals to different virtual processors.

Software in remote storage can be easily shared between bare-metal clients, making things easier for system administrators with a group of clients. For example, one virus scan at a university computer center’s shared software package at the TC server can be equivalent to independent hard disk scans for hundreds of clients.

As to the implementation of TC solutions, the prerequisite is finding a unified software-hardware interface with which software can be independently developed without having to consider the different underlying hardware platforms. This comes in the next chapter, the UEFI.
3. UEFI Introduction

On the day PC equipment was invented, Basic Input and Output System (BIOS), a kind of firmware in personal computers, was also created to work as the lowest system software with the hardware platform. In most cases, BIOS is regarded as part of the hardware, especially from a software developer’s perspective. The BIOS architecture in the computer motherboard has remained almost unchanged during the past decades (Vincent, Michael, and Suresh, 2010) (Michael et al., 2009)—still heavily bound to the hardware platform and OS customization with less abstraction, written in real mode assembly language, almost impossible to add extension code to due to IP protection and the difficulty of assembly coding, and having most features still limited to power-on and self-test (POST). To change this scenario, a new firmware interface was proposed during the nearest millennium and gradually adopted by industries as the standard.

UEFI was originally named “EFI” and designed for Itanium® platforms. The original goal of EFI was to define a relatively stable interface for OS loader before silicon and platform design was finalized (Mark, Vincent, and Michael, 2011). One of the most important UEFI characteristics—support of high-level C language interface—came into being at that time. UEFI defines the interface between the operating system and the platform firmware, and is aimed for a wide range of different hardware types, from desktops and mobile servers to embedded devices like infotainment systems, telecom equipment, and mobile phones. UEFI provides an evolutionary way to migrate from legacy. It makes the addition of BIOS extensions as simple as possible by considering firmware customization from the start. UEFI supports current industry standards like ACPI. The purpose of the UEFI interfaces is to define a common boot environment abstraction for loading UEFI images, including UEFI drivers, UEFI applications, and UEFI OS loaders, as BIOS extensions.

Figure 3 depicts the UEFI position in a computer system (i.e., between OS and firmware), and is composed of boot service, runtime service, and data tables. Data tables contain platform-related information like ACPI and SMBIOS. The boot service is used during the pre-boot period, and run-time service is for the OS stage. The underlying platform is abstracted for the coming OS boot process using these data tables and service calls.

We emphasize that UEFI is an interface specification that does not dictate how the platform should be built. The “how” is relegated to PI, the platform initialization specification. PI describes the phases of control from the reset to the success phase of operation, including a UEFI-compatible environment. Based on Intel’s sample UEFI implementation, a PI spec was published as the platform implementation guide. Figure 4 shows the internal architecture (Unified EFI Inc., 2009).

The UEFI interface separates firmware and OS. The platform firmware is composed of the PI foundation (so-called “Green-"H"), platform drivers, and silicon component modules. The Foundation is a platform-independent common code that can be used for any hardware platform. Users can add customized driver execution environment (DXE)/ pre-EFI initialization (PEI) drivers to the foundation to form the whole BIOS (explanation below).
In PI spec, the UEFI boot process is divided into a sequence of stages, as shown in Figure 5. The SEC (security) stage owns the task of basic security checking and initialization. In most implementations, SEC is merely a couple of simple instructions jumping to PEI, the pre-EFI initialization stage.

The PEI phase works when the platform is still in its nascent state, even without memory. PEI is usually responsible for the initialization of permanent memory and describing that in hand-off block (HOB) data structure, putting the firmware volume location in HOBs, and passing the control to DXE. PEI is composed of the PEI foundation and PEI modules—the former is the common core code required to run the latter. PEI has no memory, so its features can only be limited to the following: locating, validating and dispatching PEIMs, communication between PRIMs, and providing hand-off data for DXE.

Most of the system initialization is performed during the DXE stage. The components of the DXE phases are: the DXE foundation, the DXE dispatcher, and a set of DXE drivers. The DXE foundation consumes the HOB list and the services of the DXE architecture protocol to produce a set of UEFI boot services, runtime services, and DXE services. It is used to discover and execute the DXE drivers in correct order. The DXE dispatcher discovers DXE drivers stored in the firmware volume, and loads them in their proper order. The DXE drivers are used to initialize the processor, chipset, and platform components, as well as to provide SW abstractions for system services, console devices, and boot devices.

After the DXE dispatch, the boot device selection (BDS) phase takes control of the platform, working together with DXE to establish consoles and try to boot the OS. Once the OS is booting, all DXE services exit except a few run-time services. BDS is also responsible for implementing the platform boot policy, which provides a flexible mechanism for system vendors to customize the user experience.

In conclusion, UEFI is the interface between software stack (operating system) and hardware platform (firmware), and PI is the platform implementation specification. BIOS implementation following UEFI and PI specs could break the limitations of legacy BIOS by providing a unified interface without precluding opportunities for differentiation.
4. Problem Statement and the Ideal TC Implementation of UEFI

With the background for TC and UEFI given in previous chapters, we can proceed to the problem statement of TC with an ideal UEFI implementation, without consideration of hardware or software limitations. The TC working model is abstracted in Figure 6.

The bare-metal machine M, as the TC client, is connected to TC server S via the TC delivery network N. As the service bearer, the software service SW is piped as a stream to M and executed locally. Logically, SW is executed as if it was stored in M's local permanent storage. As the bare-metal, all operations including computing, memory read/write, and input/output (IO) are completed locally except for the read/write of permanent storage (e.g., hard disk).

To meet this requirement, two assumptions should be fulfilled:

- Assumption I: SW and M should be logically independent, that is, different SW can run on different M;
- Assumption II: There is no permanent storage at M, so read/write of “permanent storage” at M should be transparently completed by loading from S storage and transferring to M via N as if it was loaded at M’s virtual “permanent storage.”

Key to assumption I is the statement that the separation of software (SW) and hardware (M) is required, and a unified interface between software and hardware should be found; in other words, a unified platform abstraction to hide the details of the underlying hardware is a must. As one choice, UEFI firmware, together with the hardware as the bare-metal, provides a unified interface (UEFI) for the above operating system and applications, and a unified abstraction for the underlying hardware platform with data tables and BIOS services. This aspect alternatively explains the “transparency.”

For assumption II, however, a virtual “permanent storage” at the TC client is required. The firmware of the bare-metal should be able to hook the block IO at M from SW, forward to S, handle the block IO requests at S, send it back to M, and finally return the block IO data to SW.

To illustrate this concept, take the OS booting as an example. When the platform POST is completed, the OS boot loader takes control of the system from BIOS, and calls a sequence of disk IO to load the OS kernel to the memory. BIOS system services (for legacy OS, BIOS interrupts like int 13, and for UEFI, similar block IO system call) are called to load the data blocks (OS kernel) from “local permanent storage” into memory (Mao and Hu, 2001). These blocks are fetched from the remote server transparently. This process is what the firmware at M should do, to simulate a block-IO permanent storage with the network as if it was a real local permanent storage. After that, the OS kernel totally controls the platform.

However, care must be taken because in most mainstream operating systems like Windows and Linux, the OS does not use the BIOS call to access hardware, but rather its own device drivers. Possible reasons for this phenomenon include the following: to improve device driver performance, for the OS to take more ownership, or to support more advanced devices that are not ready at the BIOS level, for example, audio devices.

To obtain the ideal TC implementation of UEFI, the following criteria should be followed, although many of them are not aligned with any current OS or platform:

- Each UEFI service works during both boot and run-time periods, which means they do not quit even after OS has been loaded.
- All UEFI services should be optimized for performance and concurrency.
- All peripherals and functional units of the platform should be supported at the UEFI BIOS level.
- Hardware access of the boot loader should be through the UEFI service.
- The OS run-time driver should go through the UEFI service and not access hardware registers directly.

In short, UEFI, as an abstraction, should wholly separate OS and hardware, and abstract the underlying platform from power-on to power-off. The upper layer software stack, either boot loader or OS, will not directly touch hardware registers. The internal architecture is shown at A of Figure 7.
5. Implementation of TC with UEFI

Notwithstanding that the ideal TC implementation in Chapter 4 looks attractive and is quite simple in philosophy, we must return to reality and figure out how we can make the current mainstream OS work in the TC way.

The real situations, in short, our constraints, are listed below:

- Most current operating systems are legacy OS. They do not invoke the UEFI boot service call, but rather int13-like legacy BIOS calls in the boot loader.
- The OS seldom uses BIOS-level run-time system calls to interact with hardware platforms.
- Few system peripherals are supported at the BIOS level, most are devices to bring the OS up. Many devices, such as audio, are not supported before the OS is ready.
- Most UEFI BIOS are single-threaded, and code is not re-entry, which is a must for run-time drivers.
- Most UEFI device drivers are designed not for performance but compatibility.

Hence, compromises should be made to make TC work with the current available OS. Relatively close-source OS like WinXP are more difficult than open-source OS like Linux because the Linux kernel source can be changed freely.

We may look back to the requirements of low-level platforms from the implementation perspective. First is the logical interface between hardware and software to form the unified platform abstraction. Such an interface enables software stack migration across different hardware platforms—running the same software stack on different hardware and vice versa. Furthermore, TC relies on building network-based virtual permanent storage, so it would be best to not only include network interface card (NIC) support at the firmware level, but also provide mechanisms to encapsulate network service in virtual permanent storage devices. Separation or abstraction is the key, and network support is part of this issue.

Following is a look at different TC implementations using UEFI firmware in an evolutionary way, together with the analysis of each solution. These are based on the author’s work for the joint project between Intel and Tsinghua University.

Diagram B of Figure 7 is the original computer system, where the BIOS is bound to the hardware platform. In the pre-boot period, the OS loader runs on top of the firmware by invoking UEFI services. However, during run-time service, the OS directly manipulates the hardware registers. As the interface, UEFI only works during the pre-boot stage.

To solve the UEFI platform abstraction issue in the run-time period, the full virtualization solution in Diagram C comes into being, where the BIOS is customized to become a lightweight firmware-level virtual machine by adding the “value-add” module. The UEFI interface between the software stack and platform, including BIOS and value-add, is upgraded to a virtual machine-enabled UEFI interface that works not only in the pre-boot period, but also in the run-time period. The obvious improvement of the full virtualization
solution, compared with the original computer system in Diagram B, is that the guest OS does not need to touch the real hardware, but only manipulates the virtualized hardware abstraction of the virtual machine with the extended UEFI interface from pre-boot to run-time. No matter which kind of hardware is used, the “hardware” touched by the guest OS is identical—the virtual machine. The benefit of full virtualization is better platform abstraction and 100% separation of hardware and software. However, performance is a bit limited. Intel’s internal evaluation (Huang et al. 2008) of this solution is less than 20% of native performance. Moreover, certain usage scenarios heavily dependent on hardware peripherals like graphics will be impacted. Typical examples include CAD projects and animation.

The issues of full virtualization solution make us think, what if we can complete some tasks using hardware, not virtual machines? This is the partial virtualization solution in Diagram D. It only hooks the necessary hardware access in BIOS while letting other hardware access go down directly to the hardware. In our case, only block IO is virtualized, whereas other hardware operations are native. Compared with the full virtualization solution, partial virtualization greatly improves performance by using local hardware more effectively. Intel’s evaluation (Huang, 2008) shows overall performance is about 80% of native performance for most computer applications. However, another issue arises: the IO hook needs silicon virtualization support (e.g., VT) in IA processors. A brief explanation of the problem follows: the device drivers in the boot loader and OS can be abstracted by a sequence of IO instructions. We have to hook the specific range of IO instructions related to permanent storage (e.g., from 0x1f0 to 0x1f7 for IDE devices), trap to BIOS code, handle it and forward it to the remote server. For other IO instructions, we will let them pass through directly to the hardware. The instruction level hook has to be supported by CPU virtualization (Intel, 2006).

The hardware dependency of partial virtualization solution blocks its adoption in some cost-sensitive scenarios like education and low-end mobile internet devices, because many entry-level or mobile processors do not support virtualization. This aspect leads to the last solution shown in Diagram E, where we move the value-add from BIOS to OS. We replace the OS block IO driver with a new one that directly forwards the block IO to the remote server at the OS level. The obvious disadvantage, however, is the additional effort required to patch the OS boot loader and kernel.

Putting aside the architecture analysis for the different TC implementations above, the performance comparison is another interesting and potentially revealing topic. We chose OS boot time, IO performance, and typical application performance indicators for the comparison, and all test cases were run on bare-metal (Figure 7, Diagram B), non-virtualization (Figure 7, Diagram E), para-virtualization VMM (self-built BIOS payload, Figure 7, Diagram D), and full virtualization VMM (XEN, Figure 7, Diagram C), respectively. The hardware platform of the test bed was Intel Core™ 2 E7500 with 2GB memory, 300GB hard disk, and a 1000BaseTX network adapter. A clean-built Windows XP and Fedora were used as the target OS.

For the OS boot time in Figure 8 (left panel), native and non-virtualization are very close, whereas para-virtualization is longer, and full virtualization is the longest. This result aligns with our perception that VMM, either para or full, will decrease IO efficiency, whereas non-VT may not impact significantly. We can also find this outcome in the IO performance indicators in Figure 8 (middle panel). For typical compression applications like 7zip or bzip, the performance impact also happens, but is smaller than that of disk IO-intensive applications like dd at the right panel of Figure 8.
6. Case Study: Wireless-based TC Tablet Solution

The previous chapter describes the general principle of TC implementations on PCs with wired networks like Ethernet. However, mobile internet has become hot during the past several years with the massive infrastructure building of new generation mobile networks like 3G, Wi-Fi, and WiMax. The terminal of mobile internet is no longer traditional PCs only, but embedded devices like tablets, MIDs, and Netbooks. This development makes us think whether we can extend our TC solution to mobile devices in wireless networks. However, we have to regard the following new problems in this new scenario:

- Compared with wired networks, wireless bandwidth is limited and relatively unreliable.
- Mobile platforms usually have less advanced features like VT.
- Wireless terminals are easier to lose. This feature makes data security, device-user binding, and user authentication more important.

To fit the wireless network environment, a local cache was set up in the local hard disk to buffer the block IO from the remote server. Each software service can be abstracted as a base image and a delta image. The base image was put in the local cache and the delta in the remote server. Given that delta would only be a small portion of the whole software service image, most of the block IO was handled at local—this partly solved the reliability and wireless bandwidth dependency issues. We can even put the old delta image copy into the local cache if the wireless quality is too terrible, and in the worst case where wireless is unavailable, this solution can still work.

To address security issues, a “user authentication” module was added at the BIOS level. The authentication checking must be passed with every loading of the software service to prevent illegal access.

Regarding the above new issues, we will try to present an example based on the non-virtualization model in Diagram E of Figure 6, but with some enhancements to show how TC works with the current OS and hardware. Aside from the power consumption limitation that needs to be solved at the hardware level, we will see how the remaining issues can be solved here. Figure 9 is the internal architecture of our wireless TC tablet prototype.

---

Figure 9. Case Study: Wireless-based Transparent Computing Tablet —

To fit the wireless network environment, a local cache was set up in the local hard disk to buffer the block IO from the remote server. Each software service can be abstracted as a base image and a delta image. The base image was put in the local cache and the delta in the remote server. Given that delta would only be a small portion of the whole software service image, most of the block IO was handled at local—this partly solved the reliability and wireless bandwidth dependency issues. We can even put the old delta image copy into the local cache if the wireless quality is too terrible, and in the worst case where wireless is unavailable, this solution can still work.

To address security issues, a “user authentication” module was added at the BIOS level. The authentication checking must be passed with every loading of the software service to prevent illegal access.

- People care more for applications and not OS, and so may change the whole software stack at mobile internet terminal (e.g., from Meego to Android)
- As is typical in consumer markets, mobile devices see more fashion trends in terms of UI, and should support new IO devices like touch screens.
- Power consumption and performance are perennial issues for handheld devices.
access to the TC system. Furthermore, the local cache is encrypted—this feature prevents illegal users from using the device, and protects both local and remote data.

As to the selection of different OS environment, this is an inherent advantage of TC.

Most wireless-based devices are touch screen-based; thus, a graphic touch screen UI ought to be added at the BIOS level to make the pre-boot environment more user-friendly—user authentication and the selection of different software service environments are completed here.

In conclusion, the major characteristics of Figure 9 are:

- A block IO cache in local storage;
- User authentication at the BIOS level;
- Data encryption for local storage protection;
- Virtual disk management by combining same base image and different delta image to save disk space for different users, thus providing more OS boot options for end-users.

Detailed information can be found in the 2011 BJ IDF foil (Wu and Liu, 2011).

7. UEFI’s Value to TC and Summary

Based on the previous analysis and examples, we can conclude UEFI’s contributions to Transparent Computing below.

- As a general interface between OS and platform and a widely adopted industry standard, UEFI provides a unified abstraction for the underlying hardware platform.
- Modular design and flexible internal architecture make the BIOS/firmware extensible, and provide the capability of adding new modules to firmware.
- Special UEFI-based value-add modules are available at the firmware level (e.g., security and local storage management).

In terms of technology, we may not say UEFI is a required prerequisite for TC implementation. Alternative technologies like PXE-boot payload, remote file access such as NFS, and some OS-level disk IO access methods like iSCSI, might partly solve the TC problems discussed here. However UEFI is more generic and standardized, and has already been widely adopted by industries. It is easier for developers to get resources, either free sample implementations or documentations (Geeknet, online), and hence easier to be used for TC implementation.

In summary, Transparent Computing is the implementation of pervasive or ubiquitous computing. The characteristics of TC are the separation of software stack and hardware platform, as well as the separation of computing and storage. UEFI defines the interface between operating system and the platform firmware, which abstracts the underlying platform well. Additionally, UEFI makes it possible to add modules to BIOS. This aspect meets TC well. Intel has worked on UEFI-based TC solutions for years on both “PC + wired network” and “tablet + wireless network.” In the future, the following directions will be focused on to make TC available everywhere: SaaS for wireless solutions, RAS-reliability, availability and serviceability, and performance and power tuning.
References


