application aware of what portions of the application is executing by which (remote) hardware.

With the present invention, portions of the application can be executed remotely, but share system resources (file system, I/O and user interface) and interact with other program components as if they were running locally. This allows the excess hardware in a computing system to be more effectively and more frequently utilized. Fully utilizing the hardware of a heterogeneous system has benefits in both power and security. Certain types of processing is more power efficient when it takes place on specific hardware, and matching the types of processing to their optimal hardware can have a large impact on the overall power efficiency of the system. From a security standpoint, an attacker needs to know the underlying architecture of a system to crack an attack payload. If the architecture is rapidly and randomly changing, it will be difficult for the attackers to successfully match their payload to the changing architecture.

In some embodiments, a common interface layer, such as an application programming interface (API), is identified in the application to allow the transfer of the state to occur without requiring modification of the application. The present invention allows the state to be transferred and system resources (e.g., remote processing hardware) to be shared among different architectures.

FIG. **1** depicts an exemplary simplified heterogeneous processing environment, according to some embodiments of the present invention. As shown, an application **102** is being executed on a controller (local hardware) **104** having resources **118**, such as a storage medium **120**, a communication network **122**, a user interface **124**, and the like. A first portion of the application **106** may be seamlessly transmitted to a first (remote) hardware **108**, a second portion of the application **110** may be seamlessly transmitted to a second (remote) hardware **112**, and a third portion of the application **1114** may be seamlessly transmitted to a third (remote) hardware **116** to execution, while a portion of the application is being executed by the (local) hardware **104**. Once the execution of the first, second and third portions are each completed by the respective first, second and third (remote) hardwares, the results (and state) of those portions are transferred back to the controller to be integrated back to the application under execution by the controller **104**.

FIG. **2** is simplified timing diagram of a computer program, according to some embodiments of the present invention. The diagram shown is only a subset of the full timing process, showing only when architecture agnostic state is being transferred from the remote hardware to the controller. Before this would be taking place, the controller would have spawned the application on the remote hardware already. This process first identifies one or more execution boundaries. An execution boundary is the point where execution of the application is stopped on one architecture, the execution state is saved, packaged and sent to another (remote) architecture for execution. In some embodiments, this boundary needs to be a currently utilized application interface, because by choosing an interface already used by application, the application does not have to change to utilize the present invention. Possible examples of this boundary include system calls or Portable Operating System Interface (POSIX).

Once the execution boundary is understood (identified), a method of capturing execution state is identified. This method captures all necessary (execution) information to allow processing to be transferred and restarted on another system, regardless of the underlying architecture of that system.

The method of transfer between processors can be anything that provides a real time transfer capability both to and from the remote hardware, because the originator hardware may be waiting to resume processing until it hears back from the remote hardware. In this case, other applications may be running on the originator hardware at this time, so computing resources are not wasted.

The applications (or sub-applications) that will be running on the remote hardware are also identified, and transferred to their corresponding hardware. That is, identifying which applications are running where, before they are sent to, and started on their respective hardware. This can be done in advance of the execution, or on the fly, allowing the execution assignments to be done at random. The result, is an application-set executing on a distributed set of hardware, but from the applications perspective running on a single system.

FIG. **3** is an exemplary process flow for distributing execution of a computer program to a plurality of hardware architectures of different types, according to some embodiments of the present invention. As shown in block **302**, the computer program/application is analyzed to identify a plurality of execution boundaries within the application. Generally, computer programs that require an operating system to execute have predetermined points where execution control is transferred from the program to the operating system, and point where it is returned to the program. In addition, computer programs often rely on shared libraries in order to not recreate commonly used functionality. That is, there is an execution boundary when a program transfers execution control to the library, and when control is transferred back to the program. The execution boundaries that are identified will vary in how often they are used and how widespread their usage is.

FIG. **4** shows a simplified process flow for identifying or selecting a number of execution boundaries in an application, according to some embodiments of the present invention. In block **402**, the binary (the binary executable, which is the form the program will exist in on disk right before it is executed, after all build and compilation steps are completed) of the program is analyzed to determine which operating system and libraries are used. Next, in block **404**, the invention determines what execution boundaries the identified operating system and libraries utilize, within the application. Boundary information is then collected for selecting one or more execution boundaries within the application. The boundary information may include calling frequency of the boundary, number of unique calls, call timing (when in a program calls are made), calling dependency information (i.e., who makes the call, the application or shared library), and the like.

Referring back to FIG. **3**, one or more execution boundaries is (are) selected from the plurality of execution boundaries, in block **304**. In some embodiments, the execution boundary is selected based on the design goal of the system. For example, if all execution is destined for the same place, then an execution boundary where there is an execution bottleneck is chosen. Conversely, if execution is to be widely distributed, then an execution boundary that is more spread out is chosen. The boundary could also be selected based on type of application running, for example a network application would require a boundary that provides flexibility with the networking interfaces it provides. Also, a graphical application would require direct access to GPU interfaces. Another example may be data backup applications that need access to storage interfaces.

In block **306**, the computer program is linked to the selected one or more execution boundaries. In some embodiments, the linking is accomplished by placing a program in the identified execution boundary, which intercepts all communication. This interceptor program (or a sniffer program)