Bridging Functional Heterogeneity in Multicore Architectures

Dheeraj Reddy, David Koufaty, Paul Brett, Scott Hahn
{dheeraj.reddy,david.a.koufaty,paul.brett,scott.hahn}@intel.com
Intel Corporation

ABSTRACT

Heterogeneous processors that mix big high performance cores with small low power cores promise excellent single–threaded performance coupled with high multi–threaded throughput and higher performance–per–watt. A significant portion of the commercial multicore heterogeneous processors are likely to have a common instruction set architecture(ISA). However, due to limited design resources and goals, each core is likely to contain ISA extensions not yet implemented in the other core. Therefore, such heterogeneous processors will have inherent functional asymmetry at the ISA level and face significant software challenges. This paper analyzes the software challenges to the operating system and the application layer software on a heterogeneous system with functional asymmetry, where the ISA of the small and big cores overlaps. We look at the widely deployed Intel® Architecture and propose solutions to the software challenges that arise when a heterogeneous processor is designed around it. We broadly categorize functional asymmetries into those that can be exposed to application software and those that should be handled by system software. While one can argue that new software written should be heterogeneity–aware, it is important that we find ways in which legacy software can extract the best performance from heterogeneous multicore systems.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management; C.1.3 [**Processor Architectures**]: Other Architecture Styles - Heterogeneous (hybrid) systems

General Terms

Algorithms, Performance

Keywords

Functional Heterogeneity, Multicore, Shared Asymmetric ISA, Intel® Architecture, Operating systems

1. INTRODUCTION

Advances in semiconductor technology have enabled processor manufacturers to integrate more and more cores on a chip. Most commercial multi-core processors consist of identical cores, where each core implements sophisticated micro-architecture techniques, such as super-scalar and out–of–order execution, to achieve high single–thread performance. This approach can incur high energy costs as the number of cores continues to grow. Alternatively, a processor can contain many simple, low–power cores, possibly with in–order execution. This approach, however, sacrifices single–thread performance and benefits only applications with thread–level parallelism.

A heterogeneous system integrates a mix of *big* and *small* cores, and thus can potentially achieve the benefits of both [1, 7, 9, 15, 16, 27, 29]. Despite their significant benefits in power and performance, heterogeneous architectures pose significant challenges to operating system design [6], which has traditionally assumed homogeneous hardware.

Heterogeneous architectures can be broadly classified into two classes that are not mutually exclusive: *functional asymmetry* and *performance asymmetry*. Functional asymmetry refers to architectures where cores have different or overlapping instructions set architecture (ISA). For example, some cores may be general–purpose while others perform fixed functions such as encryption and decryption. Performance asymmetry refers to architectures where cores differ in performance (and power) due to differences in micro-architecture or frequency. Within the functional asymmetry design space, two trends are likely to exist: the ISA might be non–overlapping (such as the case of the Cell processor [24] or hybrid CPU/GPU designs [22]) or the ISA might be overlapping.

We believe a significant portion of future commercial multicore heterogeneous processors are likely to have a common ISA. However, due to limited design resources or micro-architectural goals, the big core is likely to contain ISA extensions not yet implemented in the small core. Therefore, such heterogeneous processors will have inherent functional asymmetry at the ISA level and face significant software challenges. This paper analyzes the software challenges to the operating system and the application layer software on a heterogeneous system with functional asymmetry, where the small core ISA overlaps with the big core ISA as in Figure 1-(a) or one is a proper subset of the other as in Figure 1-(b). We find that software has several options to leverage heterogeneity. However, each option is influenced greatly by the design goals of a particular software stack.

Our paper is novel in several ways. First, previous work on heterogeneous architectures has focused on the performance advantages of designing performance asymmetric systems assuming a homogeneous instruction set. This is very unlikely since small processors stand to gain power/performance advantages by sacrificing rarely used hardware or adding

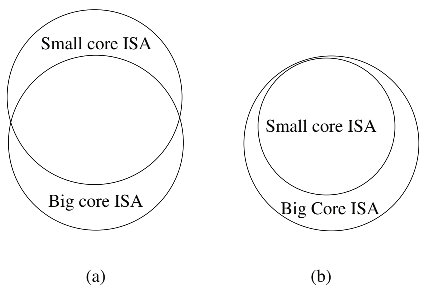


Figure 1: Overlap of instruction set architecture (ISA). In (a) the two cores share a common ISA, but each has special purpose features. In (b) one core ISA is a proper subset of the other core.

hardware accelerators. Additionally, processor manufacturers are constrained by their design resources and will try to reuse existing core designs with their exiting ISA as much as possible. Second, previous work on functional asymmetry has been limited to kernel support for user-level ISA. To our knowledge, ours is the first comprehensive analysis of the diverse heterogeneity challenges faced by the operating system, runtime and user space and a systematic methodology to bridge the gap between the different ISA.

The rest of this paper is organized as follows. Section 2 describe the different design alternatives to support heterogeneity on each feature set. Section 3 enumerates the list of most common functional gaps that need to be addressed, either with software or a combination of software/hardware. It also establishes a baseline of must have features across architectures. Section 4 describes two hardware prototypes used to implement some of the ideas presented. Section 5 discusses how to address some of the implicit assumptions about homogeneity made by operating systems that break in the presence of heterogeneous systems. Section 6 discusses related work and Section 7 presents our conclusions.

2. DESIGN SPACE

Heterogeneous multicore architectures present new challenges in designing both the hardware – operating system and operating system– application interfaces: (1) how should the operating system exploit the heterogeneity — are the heterogeneous capabilities of sufficient value that the operating system should always use them (even if this requires migrating execution to other core types), should the feature be used opportunistically if available, or should the feature be ignored completely? and (2) how should the operating system expose the heterogeneity to applications — should it present the illusion of a homogeneous system (either with or without the heterogeneous feature) or should it expose the heterogeneity, requiring applications to cope with the resulting asymmetry?

These questions must be answered independently for each heterogeneous feature, weighing the potential functional and performance benefits with the added complexity in the operating system (which must manage the demand on resources and provide sharing mechanisms, security etc) and applications. Choices are often constrained by hardware implementation issues which limit flexibility and existing operating system design choices which favor different software calling conventions (system API, device driver, IOCTL etc).

Model	*User Space View*	*OS Support*
Restricted	Homogeneous	De-featuring
Hybrid	Heterogeneous	Hetero-API extensions
Unified	Homogeneous	Feature unification

Figure 2: Overview of the solution space. The operating system must make choices of what features it exploits and what features it exposes to user level.

In addressing these key questions, a number of common architectural strategies have emerged each of which can be applied independently to different features. Figure 2 lists these solutions and their impact on the operating system and user space. In the next sections, we examine each of these approaches in detail.

2.1 The Restricted Model

The Restricted Model leverages our assumption that future heterogeneous multi-core processors share a significant portion of their ISA, sufficient to provide a complete execution environment to system software. The restricted model targets software at a hypothetical core which implements only this shared ISA.

An operating system which chose to use the restricted model internally for the handling of a heterogeneous feature would be implemented to utilize only the common ISA uniformly provided on all the cores. Independent of this choice, an operating system could choose to expose the restricted model to higher level software by hiding (preventing enumeration), disabling (preventing usage) or forbidding (by convention) the asymmetric ISA features from higher level software.

Implementating the restricted model presents a number of challenges:

- There must exist some useful, common subset of the ISA on all cores in a system. Hence the restricted model cannot be implemented where each core type presents similar but incompatible implementations of a feature.
- The common ISA features, which is specific to the combination of both cores, must have been identified at operating system design time, significantly increasing the software enabling costs of any new heterogeneous platform.
- The common subset of the ISA represents a new architecture, similar to but independent from either of the core architectures, onto which system software will need to be ported and validated.
- System software will not be able to utilize the benefit of the heterogeneous features of the platform, thereby reducing the benefit gained by adding heterogeneity to the platform.

- Additional hardware support may be required in the platform to prevent the enumeration or usage of heterogeneous features which require OS support for configuration, configuration or state management from third party applications.

2.2 The Hybrid Model

In the cases where no common ISA exists, or the heterogeneity exposed by the platform provides sufficient benefit to software, the operating system may choose to explicitly implement and/or expose the asymmetry available in the platform. The Hybrid Model requires potentially significant software modifications to gain the benefits of the heterogeneous platform.

In the case where the operating system explicitly addresses heterogeneity, the operating system can be divided into core specific and core agnostic portions. Core specific code within the operating system, for example interrupt handlers and scheduler, will never migrate and hence can utilize dedicated data structure and code paths to manage and benefit from the heterogeneous features. Core agnostic code with the OS is typically preemptable and can be migrated by the CPU scheduler, and can be enabled using the same techniques as heterogeneous aware applications.

Heterogeneous software must first interrogate the system to establish the heterogeneity available in the platform, for example by querying a new system database or testing each of the available cores (Figure 3). The existing core affinity capabilities presented by operating systems can then be used to enable the application to adapt to the heterogeneous environment. However, heterogeneous applications utilizing core affinity must balance the cost of using core affinity with the benefit gained from the heterogeneous feature. Selectively executing code optimized for the current processors (Figure 4) incurs a potentially significant system call overhead to set and clear the affinity mask (measurements on an Intel® Xeon® Processor E5450 processor running Linux 2.6.27 show this overhead can be as 5000 cycles). This cost can be amortized over many instructions by forcing the thread to remain on the big cores (Figure 5), at the risk of overloading these cores with threads which cannot be migrated.

Implementation of such modifications may require heterogeneous aware compilers, multi-path libraries, or ”fat binaries” containing multiple optimized versions of applications requiring the developer to select independent algorithms optimized for the platform asymmetry.

This diversity in implementations creates a resource management issue for an operating system– applications which are executing algorithms optimized for one type of core may not be able to execute efficiently (or at all) on the other cores. Heterogeneous applications can express these dependencies using CPU affinity, but at the cost of limiting OS load balancing – impacting the ability of the scheduler to optimize system throughput.

2.3 The Unified Model

The Unified Model attempts to hide the resource management issues of exposing heterogeneity by presenting the super–

```
// test_capability tests for SSE 4.1
int test_capability(void) {
    unsigned int eax, ebx, ecx, edx;
    eax = 1;
    asm("cpuid" : "=a" (eax), "=b" (ebx),
                  "=c" (ecx), "=d" (edx)
        : "0" (eax), "2" (ecx));
    return (ecx & 1<<19);
}

cpu_set_t discovery() {

    cpu_set_t current, capable;
    int CPUS = sysconf(_SC_NPROCESSORS_CONF);
    unsigned int eax,ebx,ecx,edx;
    int i;

    CPU_ZERO(&capable);
    for(i=0;i<CPUS;i++) {

        // set_affinity to cpu i
        CPU_ZERO(&current);
        CPU_SET(i, &current);
        if(!sched_setaffinity( 0,
                sizeof(cpu_set_t), &current)) {
            // if the cpu has capability
            if(test_capability()) {
                CPU_SET( i, &capable );
            }
        }
    }
}
```

Figure 3: Discovering SSE4.1 in a Linux application

```
cpu_set_t previous, current;

sched_getaffinity( 0, sizeof(cpu_set_t), &previous);
// pin the task to the current core
CPU_ZERO(&current);
CPU_SET(sched_getcpu(), &current);
if(!sched_setaffinity( 0,
        sizeof(cpu_set_t), &current) &&
   test_capability()) {
    // use SSE 4.1
} else {
    // use alternate algorithm
}
sched_setaffinity( 0, sizeof(cpu_set_t), &previous);
```

Figure 4: Linux application code to utilize SSE4.1 only if it is present on the current CPU

```
// early on in the code, we build maps of all and
// SSE4.1 capable processors
cpu_set_t default;
cpu_set_t capable = discovery();
sched_getaffinity(0, sizeof(cpu_set_t), &default);

// at each use of SSE4.1 instructions, we set the
// CPU mask, possibly forcing process migration
sched_setaffinity(0, sizeof(cpu_set_t), &capable);

// use SSE 4.1
...

// finally, restore default affinity to maximize
// throughput
sched_setaffinity( 0, sizeof(cpu_set_t), &default);
```

Figure 5: Linux application code to utilize SSE4.1 instructions, forcing migration if required

set of capabilities of the cores to software.

Capabilities present in any of the cores in the architecture are exposed to software as if they are present on all the cores in the system, with the operating system providing transparent services to ensure the correct functioning of software which uses features not present on the current core.

When a process attempts to utilize a capability not present on the current core, the core transfers control into the operating system fault handler. For example, software which executes an instruction on a core which does not support that instruction generates an illegal instruction fault. Operating systems designed for homogeneous systems will normally interpret this fault condition as a fatal error and terminate the process (for example Linux will deliver a SIGILL to the user process). However, on a heterogeneous systems the fault may have occurred because the process executed an instruction which is only present on some subset of the available cores. With the unified model, the operating system can use one of a number of strategies to continue execution of the faulting process, namely, process migration, instruction emulation and instruction proxying.

2.3.1 Process Migration

The operating system migrates the process to another core type and re–executes the faulting instruction, on the assumption that this core will support the missing capability. Processes which fault at the same instruction address of each core type are handled in the normal way for a homogeneous system. This approach was introduced in [18] and it is referred as *fault-and-migrate*.

Process migration provides a simple mechanism to present the unified model to software, however if many applications attempt to use heterogeneous capabilities present on only a few cores, process migration may the system to become imbalanced. Using load balancing to migrate heterogeneous processes back onto the other cores could result in excessive process migrations, as processes bounce between cores. Finally, if the ISA of one core type is not a strict subset of the other core type, it is possible for pathological applications to bounce between cores as alternate instructions may force migrations.

2.3.2 Instruction Emulation

As an alternative to migrating the faulting process, an operating system could choose to emulate the faulting instruction. Early Intel® Architecture system included instructions for floating point operations which were only present if an 8087 math coprocessor was installed in the system. operating system such as Linux provided an instruction emulator for the 8087 for systems which did not include the hardware. However, software emulation is significantly slower than hardware not just because the dedicated hardware accelerators are not present, but also because a fault and decode operation must be taken for each emulated instruction. Hypervisors face a similar issue with emulation of privileged instruction sequences, and have developed techniques to reduce (but not eliminate) the performance overhead. Xen [3] uses the QEMU [5] emulator to decode streams of many instructions, reducing the number trips into the fault handler. VMware [25] uses binary translation with aggressive caching and pinhole optimization to achieve a similar result.

2.3.3 Instruction Proxying

Finally, the operating system could encapsulate the faulting instruction in a lightweight execution context and pass it to the other core type to be executed. Executing only the faulting instruction on the other core type can allow the process to gain the performance benefit of process migration with a reduced risk of over–committing the more capable core types. However, encapsulating the faulting instructions context is complex and expensive. Each operation will incur a significant performance penalty because of the inter–core communication required, not just on the receiving core which must perform the work but also on the sending core since it will be difficult to schedule other work in the short period in which it would be waiting for completion of the instruction. For any instruction sequence involving memory operations, the core will have to change memory contexts to execute the instruction, impacting cached data.

2.3.4 Common Problems

Although the unified model provides a powerful tool for legacy application support, each technique presents significant problems since they require that each core type must generate an exception for every feature that it does implement. Features which presents a common interface but provides alternate definitions would therefore be problematic to implement. Additionally, the techniques required to implement the unified model introduce significant performance overheads and both *Process Migration* or *Instruction Proxying* add intercore dependencies which could result in faults or deadlocks in performance critical sections such as fault handlers and device drivers. Since operating system changes to address platform heterogeneity would be required in order to expose or exploit the unified model, we believe the benefit to the operating system of the unified model is limited and it is much more likely that a combination of the restricted model and the hybrid model would be used for operating system code.

3. FUNCTIONAL ASYMMETRY

Bridging the gap between instruction sets on heterogeneous systems requires both hardware and software support, including policies for the heterogeneous model in effect for each asymmetric core feature. In this section, we will describe how various functional asymmetry issues affect the operating system and give examples on Intel® Architecture platforms. While this is not an exhaustive list, it showcases some of the most common challenges to bring up the operating system components that depend on the hardware features, including feature enumeration, memory models, and instruction set.

3.1 Feature enumeration

Enumerating the features supported by each core and adopting a model to use and expose those features to applications is perhaps the most critical support needed for a true heterogeneous–aware operating system. Two key issues need to be addressed. As described in Section 2, the operating system must decide what heterogeneous features it will support and which ones will be exposed to user space.

3.1.1 Core features

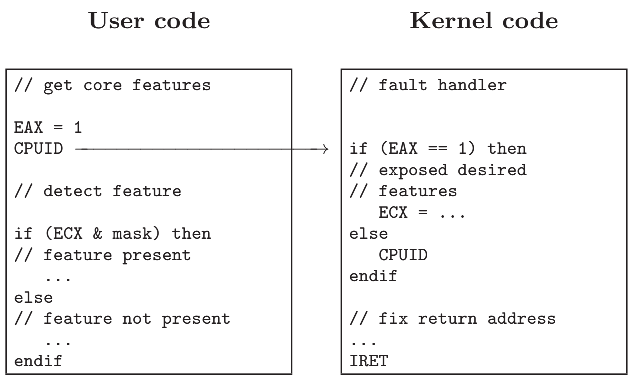
Traditionally, operating systems have been designed for homogeneous hardware and would check feature availability by checking it in one processor and assuming it exists everywhere else. Examples of this are instruction subsets, architectural registers and operating modes such as virtualization. Feature enumeration must be fixed in one of three ways. First, for features that will be supported only if they exist in every core (the restricted model), the feature check need to use a global set of common features instead of a core–specific feature support. Second, for features that will be supported asymmetrically (the hybrid model), most data structures would need to be made per–core to guarantee only support in the current core is checked. Moreover, initialization of such features made in one processor (for example, the boot processor) will require awareness the feature availability in other cores. Finally, features that would be supported transparently regardless of hardware support (the unified model) require an approach similar to the hybrid model, with the addition of software and hardware support described next.

3.1.2 Exposing features to user space

An operating system must decide what kind of asymmetry, if any, it wants to expose to user space. This is important because libraries and code that are not asymmetry–aware could break. For example, if a library checks for the existence of a certain high performance feature that only exists in a subset of the cores it could lead to lower performance when the feature is checked in a core without it. Even worse, if the feature is detected, but the thread is later migrated to another core without it, it could lead to a fault. In section 3.4 we discuss a potential solution for this issue.

It is not practical to predict what policies general purpose operating system will support. For this reason, we propose that hardware provide a mechanism to allow software to select which features will be exposed to user level on each core. This could be implemented in hardware by intercepting those instructions used for feature enumeration executed in user space, similar to the techniques used in virtualization [12]. In the Intel® Architecture, the *CPUID* instruction is used for feature enumeration, so one alternative is is to provide an exception when user code executes the *CPUID* instruction and let the operating system generate the appropriate result for each feature in the fault handler. Figure 6 shows an example implementation. Using this mechanism to implement a restricted or unified model, user code that uses *CPUID* to detect features does not have to be modified, while the kernel code fault handler provides the set of features that it wants to expose to user level. Alternatively, hardware can provide overrides that allow the operating system to selectively disable features that it does not want to expose to user level, easing the implementation of The Restricted Model.

The method above is generic enough to allow flexibility to support many models. But any other methods that allows the operating system to mask out the existence of a hardware feature would suffice.



```
// get core features

EAX = 1
CPUID

// detect feature

if (ECX & mask) then
// feature present
   ...
else
// feature not present
   ...
endif
```

```
// fault handler


if (EAX == 1) then
// exposed desired
// features
   ECX = ...
else
   CPUID
endif

// fix return address
...
IRET
```

Figure 6: Exposing a desired target instruction set to user space.

3.1.3 Non–architectural features

Some cores provide features that are model–specific and therefore non–architectural. Such features need to be addressed in software. For example, the address, bits and encoding of non–architectural model–specific registers in the Intel® Architecture can vary across cores. Its use in the kernel should be conditional of the core type where the kernel is executing.

3.2 Memory and interrupt domains

One of the fundamental principles in which most traditional operating systems rely on is the ability of all processors to share a single physical memory address space and interrupt domain. Although some non–traditional operating systems [4] and clustering operating systems [2, 20] exists that do not require a single domain across the system, they still assume a single domain within each node.

A single memory domain requires the system to provide coherent access to all memory in the system to each core. A single interrupt domain requires the system the ability to do symmetric inter–processor communication.

Our work assumes that the hardware provides both. Therefore, all memory in the system must be cache coherent and shared. Similarly, internal or external interrupt controllers must support addressing any core in the system. One of the consequences of this is that processors that support the advanced *x2APIC* architecture of Intel® Architecture processors cannot be mixed with processors that only support the incompatible *xAPIC* architecture, unless the *x2APIC* is only run in compatibility mode (i.e. *xAPIC* mode).

One consequence of the single memory domain is that if the cores do not support the same physical address width, the operating system is limited to use the lowest physical address width of all the cores in the system for general purpose memory management. The remaining physical memory beyond this limit would be unused or dedicated to special purposes.

3.3 Virtual memory and paging

There are many hardware features that control the operation of the virtual memory and paging subsystems in the operating system. For the discussion convenience we subdivide them into three groups: page tables, paging cache structures, and cache control.

3.3.1 Page tables

Page tables control the translation between virtual address and physical address, with different paging modes associated with differences in page size and attributes. The operating system expects all processors to support the same paging modes on all cores or, alternatively, it will use only the common paging modes across all cores. While there could be a performance or functional advantage in using a paging mode only available in certain cores (for example a large page size or a no–execute bit), these page tables might be shared by threads executing on different cores types and the cost of remapping them out weights its benefits.

Given that the paging modes are the same and that all memory should be shared, it follows that the virtual address space width should be the same across all cores.

3.3.2 Paging structure caches

The design of paging structure caches such as translation lookaside buffers (*TLB*) are not visible architecturally, therefore any differences on their sizes or associativity would only have a performance impact and are irrelevant to the functionality of the operating system. However, they might have an impact on the design of the operating system scheduler [14].

Address space identifiers (ASID) are commonly used in many architectures to allow selective flushing of TLB and improve performance when switching VM contexts. Support for ASID in all cores is optional, as the operating system can either forgo ASID support completely or maintain ASID in software and only use them in cores that support them.

3.3.3 Cache control

Cache control refers to all features that control the cacheability of data, including memory types and cache line size. Significant effort is made by the operating system and application software to avoid cache line alignment issues and minimize the amount of false sharing. Additionally, software that depends on a certain cache line size to flush data out of the cache (e.g. *CLFLUSH*) might not behave properly when the cache line size varies across cores. For this reasons it is recommended that the line size be the same across all cores.

The memory type can be controlled by different structures in hardware. In the Intel$^{\circledR}$ Architecture, memory type is derived from a combination of memory type in the page tables and the memory type range registers (*MTRR*). Software expects that the memory type resolution and visible behavior of the resulting memory type be exactly the same across all cores. Without it, software would invariably use additional memory fences or stronger memory types to enforce the desired behavior in hardware, leading to a loss of performance.

3.4 Instruction asymmetry

There are two types of instruction asymmetry relevant for heterogeneous systems: the overlapping instructions that exists on all cores and the non–overlapping instructions that only exists on a subset of cores.

3.4.1 Overlapping instructions

In order to simplify the programming mode, instructions that exists in two or more cores must have identical encoding and result in identical state change. Otherwise, the whole software stack including compilers, operating systems, runtime and applications would be hampered by the non-deterministic behavior of potentially each instruction. While runtime checks for core-specific behavior can be done, they are not a practical solution since it would have to be done atomically before code path that uses instructions that are not identical.

While guaranteeing identical encoding and results is straight forward for integer arithmetic instructions, it might pose more challenges to floating point arithmetic. We expect cores to support the same precision and rounding modes.

3.4.2 Non–overlapping instructions

The instruction set exposed by each core is likely to differ and result in each core type having specialized or advanced instructions not implemented on other cores. While it is desirable that user level code executes only instructions available in the core it is executing on, it might be difficult to achieve this, particularly for legacy software. Indeed, executing an instruction that does not exist in one core can be transparently handled by the kernel by performing a fault and migrate as described in Section 2.3.1. While this is not generally desirable for high performance computing, it constitutes as last line of defense for correct execution.

There are two types of instructions that might exists only in some cores. The first type is those that operate on architectural state already present in every core. For example, SSE4 instructions [11] operate on the same *XMM* registers that previous SSE instructions do. In this case it is enough to migrate the thread using the existing context switching method. The second type consists of those instructions that operate on new state only available on certain cores. This new challenge requires changing the context switch code in two ways:

- The save area on the task structure must be able to accommodate all the state across cores.
- Context switching should only save and restore the portions of the state that is relevant to the cores it is operating on.

One example of how this can be done using high performance context switching is depicted in figure 7. On the Intel$^{\circledR}$ Architecture, the *XSAVE* instruction [11] allows saving of base and extended architectural state. There are several possibilities on how to implement this depending on the characteristics of the extended states implemented on each core. One approach would be for each core to be aware of the architectural state implemented by the other cores and design the save area format accordingly, as allowed by the definition of the save area. The operating system would set the save masks appropriately to only save the state applicable to each core. Another solution shown in figure 7 combines one core with no extended state with one core with extended state. Using the compatibility of the legacy save area, the

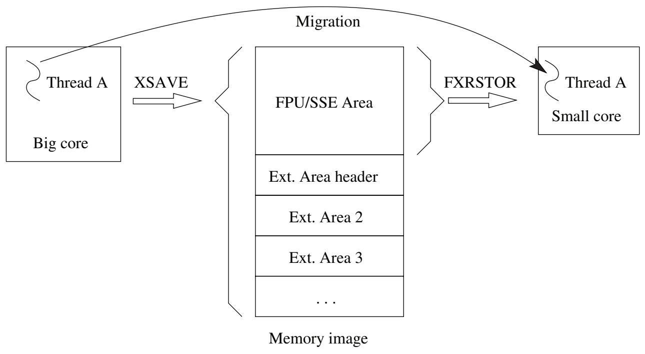


Figure 7: Mixing state save instruction with XSAVE and FXSAVE.

core without extended state can use the older *FXSAVE* instruction, while the core with extended state can use the newer *XSAVE* instruction.

Of course, as previously stated this is not a high performance solution. It is desirable that code that is not asymmetry aware be either restricted to the right core type or limited to use the common instruction set. This will avoid the performance impact from frequent migrations and the unnecessary over subscription of certain core types.

In the case of kernel code, it should refrain from using instructions that are not available in the current core (using conditional code) or simply use the common set of instructions across all cores. Support like fault and migrate is not feasible in kernel mode for two reasons. First, certain code paths that are non–preemptible cannot be transparently migrated. For example, if the code assumes that the local core's runqueue is locked, this assumption would be broken if the thread migrated. Second, even if the code is preemptible, the faulting instruction might have been intended to change the privileged state of the local core. Migrating it would incorrectly made that state change on a different core. Finally, it is hard to envision a situation where the performance benefits of using fault and migrate for kernel code would outweigh the complexity of addressing the previous issues.

3.5 System topology

Today's systems include a variety of topologies that are handled by the operating system. Examples of this are cache hierarchies, memory subsystems and non–uniform memory access (NUMA), core to socket mapping, and simultaneous multithreading.

The operating system must address the potential of asymmetric configurations. For example, the number of cores per socket could be different on each socket, or the size and number of sharers of a cache. This does not represent a particular challenge since it is already likely the operating system already using hierarchical data structures for this purpose.

One notable aspect of how the operating system deals with system topology is scheduling. When cache and memory hierarchies differ, care must be take to create scheduling domains that reflect the differences in cache topologies among the cores and sockets. Similarly, scheduling optimizations for simultaneous multithreading [23] should be only applied to those cores that support it. Finally, scheduling optimizations can exploit the diversity in the workload to improve system performance [26, 14].

3.6 Timing infrastructure

There are many platform counters that can be used for timing purposes. As long as they are globally shared in the platform they do not represent a challenge in heterogeneous systems. Modern processors such as those from the Intel® Architecture family provide a high resolution counter called the time stamp counter (*TSC*) that is increment every clock cycle. Since the clock cycle is the minimum unit of time, the TSC provides the highest resolution counter available in the platform. Additionally, the TSC can be read fast, usually orders of magnitude faster than other platform counters.

In order to support the continued use of TSC for timing purposes, hardware must provide guarantees that the TSC is incremented at a constant rate across all cores on the system. For heterogeneous systems this implies that it is desirable to have the TSC increment at the same constant frequency across all cores, regardless of the actual frequency of the core. Moreover, the TSC should not be affected by sleep or frequency events. Otherwise, the operating system will not use TSC and obtain timing information from other slower sources.

3.7 RAS and debug

Reliability, availability and serviceability (RAS) features are often critical in certain market segments. As such, the requirements on RAS features will likely be dictated by the design goals of the CPU. While software does not require the RAS features to be identical across cores, its usefulness is limited by the lowest common feature set. Supporting asymmetry on other features is possible, such as limiting the propagation of certain errors, and could be enabled only on cores that support it.

Debugging of heterogeneous systems extends well beyond the operating system kernel. While the hardware infrastructure for debug is likely to offer some advanced features in certain cores, it is expected that most basic functionality will be common across all cores to enable debuggers. Debuggers and other tools will need to be updated to reflect the asymmetry in the cores, particularly the non–common state and decoding/disassembly of non–overlapping instructions.

3.8 Performance monitoring

Most modern CPU include some fundamental infrastructure to collect performance counter that enable developers and other users to understand the application behavior. Many tools such as Intel® VTune rely on the existence of these counters to debug applications.

There are two types of performance counters in the Intel® Architecture. First, the architectural performance events are those that behave consistently across microarchitectures. Intel® Architecture implements several versions of architectural events, and it would be desirable to all cores to support the same version. Second, non–architectural events are those that are specific to the core. Due to the varying microarchitectures, non–architectural events would likely differ significantly between the cores. All layers of the software stack, including the kernel, would need to be aware of this asymmetry. Tools such as VTune would need to be updated and provide the user with proper interfaces to collect the proper events on each core.

3.9 Virtualization

If the virtualization features are not identical across cores, a VMM cannot effectively use those features. For example, it is not worth the effort for a VMM to support two guest to physical memory translation algorithms and data structures simultaneously, say a page table shadowing and a guest to physical page table, to be used conditionally on the presence of support for extended page tables on the current core.

Therefore, the VMM is likely to use only the common set of virtualization features. Yet, this by itself is complicated as some virtualization architectures such as Intel's VMX hide the memory layout of the virtual machine control structure (VMCS) from software. This format will change as new virtualization features are added and new per–VM state is saved in the VMCS, leading to the introduction of the VMCS revision id. Hence, the format and revision id of the VMCS might be different across cores, and one core would not load a VMCS with a different revision id.

Two solutions are possible. The desired one is for hardware to provide the same VMCS format and revision id, even if some fields are deprecated in some cores due to unimplemented features. The other approach, much less desirable, requires the VMM to do the translation during a VM migration across cores of different types. For example, the VMCS can be saved to a format–independent memory region using VMCS reads and saved in a new format using VMCS writes on the new core. This process adds to the cost of a VM migration, but could be mitigated by reducing the number of such migrations, for example by creating processor pools by core type.

3.10 Summary

Table 1 summarizes the discussion of the previous sections. Adoption of heterogeneous systems will require hardware and software co–development. Whilst the operating system can support some degree of functional asymmetry, it requires hardware to provide a basic subset of a common ISA, including single memory and interrupt domains, identical memory types and identical behavior for common instructions. Software, on the other side, can bridge the gap in functionality for certain classes of functional asymmetry, including core feature set, topology and non–overlapping instructions. However, in order to fully exploit performance of asymmetric systems both the operating system and the user–level layer need to be aware of it, including libraries, debuggers and performance monitoring tools.

In some cases, software is limited in the amount of asymmetry that it can tolerate or exploit. For instance, asymmetric paging modes or virtualization features, would require the additional complexity of multiple kernel algorithms and state migration between them, so it is unlikely that the operating system itself would use these features. On the other hard, applications can always deal with asymmetric features, either by explicitly programming to the hybrid model or by hiding the asymmetric features in the operating system using the restricted model or the unified model, the later particularly useful for legacy applications.

4. HARDWARE PROTOTYPES

To identify and further investigate the operating system issues that might occur in a heterogeneous systems, we experimented with two hardware platforms. The first prototype consisted of processors from different microarchitecture families in a single system. The second prototype uses identical processors but with certain features disabled in selected cores.

4.1 Multi-family platform

This platform consists of a dual socket system featuring processors from different processor families, a dual-core Intel® Atom™ Processor N330 and a quad-core Intel® Xeon® Processor E5450. This platform thus gives us a pairing of a set of high performance big cores (E5450) with a set of power efficient small cores (N330). The small cores are an order of magnitude more power efficient than the big cores but have significantly less performance to that of the big cores. We verified the raw integer performance difference between the cores using a set of microbenchmarks. While these big cores do not support any form of SMT, the small cores are two-way SMT cores. This asymmetric core prototype has significant architectural, topological as well as performance asymmetry. These systems have a modified firmware that supports the different core types. These platforms also have been modified such that an N330 core is always the bootstrap processor (BSP). We disabled the use of TSC as a clocksource since the two processors were running at different clock frequency. Thus, the HPET was the only clocksource in our operating system. This makes sure that the core with a smaller feature-set is always the smaller core. Though there is no such requirement for a heterogeneous system, it makes our platform bring–up process relatively easy. Our modified Linux kernels and the software stack above could run various performance-oriented algorithms previously proposed [17, 14] for a performance asymmetric heterogeneous systems.

4.2 Defeatured platform

The second platform consists of a system featuring the 2011 2nd Generation Intel® Core® Processor codenamed "*Sandy Bridge*". These cores come with the Intel Advanced Vector Extensions (AVX), a new 256-bit SIMD instruction set that accelerates floating point intensive applications. The AVX instructions depend on the presence of the *XSAVE/XRSTR* instructions to save and restore the extended AVX state. Using proprietary tools, we disabled the *XSAVE/XRSTR* and, therefore, AVX instructions in a subset of the cores that we designate as small cores. This provides for an architectural asymmetry in the the heterogeneous platform but almost no performance asymmetry. Applications which take advantage of the newer AVX instructions will be able to run only on the big cores. They will experience illegal–instruction exceptions if they run on the small cores. To overcome this heterogeneity, we resorted to The unified model as discussed in Section 2.3. We created the floating point state for all the

Topic	Subtopic	Hardware	Software
Domains	Memory	Same	
	Interrupt	Same	
Features	Core features	Expose to SW	Manage
	User space features	Add HW support	Policy
Memory	Page tables		The Restricted Model
	Physical address		The Restricted Model
	ASID		The Restricted Model
	Memory types	Same	
	Cache line size	Same	
	Paging caches		Transparent
Instructions	Overlapping	Same	
	Non–overlapping		Policy & manage
Topology	Cache, NUMA, SMT		Manage
Timing	TSC frequency	Same desirable	Use if same
RAS			The Restricted Model, others optional
Debug		Basic features same	Impact to tools
Performance monitoring	Architectural	Same	
	Non–architectural		Manage in tools
Virtualization	Features		The Restricted Model
	Revision id	Same desirable	Manage

Table 1: Summary of hardware and software requirements, details are given in the respective sections. Blank entries mean that there are no requirements, while The Restricted Model refers to support only for the subset of the feature[s] common to all cores in the system.

processes corresponding to the big core feature set. Thus, each process will have floating point state which assumes that it will run on the big core. Obviously, when the process runs on the small core, the floating point instructions operate on a subset of this state. AVX instructions on the other hand, operate on the complete state. To verify that our architectural hypothesis is indeed reflected in software behavior, we hand–crafted a few sample applications which use AVX instructions that exhibited the fault–and–migrate behavior and could exploit the heterogeneity using the techniques mentioned in Sections 2.2 and 2.3. We also verified that migrating legacy applications (SPEC 2006 benchmarks) across the cores of this heterogeneous systems does not result in any functional deviation or incorrect results.

5. OVERCOMING HOMOGENEITY

System software running on current multicore architectures make several implicit assumptions about the homogeneous nature of the platform. In many cases, these assumptions prevent the basic bootstrapping of the operating system itself on heterogeneous systems. However, a larger problem occurs when core algorithms such as those that govern resource allocation assume homogeneity. This results in silent suboptimal behavior of the system and in many cases, outright incorrect behavior. In this section we describe several such assumptions that we encountered while running modern operating system stacks on the two custom heterogeneous systems that were described in Section 4. Even though we focus the discussion mostly on the Linux® operating system, we believe that most of these issues are also present in other operating systems. For example, our analysis of the FreeBSD® software stack reveals that the issues we discuss below also apply to it.

5.1 CPU feature flags

Early in the boot process, the Linux kernel discovers the features that an CPU supports. However, there is only one such persistent data–structure that stores the flags of the bootstrap processor (BSP). These *capability bits* are then logically ANDed with the features that are discovered when the application processors (AP) are enumerated. Subsequently, whenever the kernel software needs to check for a particular feature flag, it refers to these common *capability bits*. While this ensures that software that assumes homogeneity will work in the kernel, it was designed to overcome minor differences in processor steppings and only supports a The Restricted Model kernel, thus sacrificing the advantages that the big cores may potentially provide.

On user space, most application software written to take advantage of advanced processor features depends on native instructions (such as *CPUID*) to discover processor features. While this is not a problem on homogeneous systems, when the operating system chooses to present a The Hybrid Model to user space, it can cause software to underutilize the system or generate exceptions. While solutions like fault and migrate (Section 2.3.1) handle these issues as a last resort method, feature detection on heterogeneous systems should be rewritten to detect per-cpu features using either native methods or new software interfaces that the application layer can count on.

5.2 Runtime code patching

One of the ways in which the Linux kernel takes advantage of the variations and improvements in hardware is by means of the "alternatives" mechanism. This allows the kernel to optimize itself at boot–time when it exactly knows what platform it is running on. During build–time, the kernel stores two (or more) implementations of a given function-

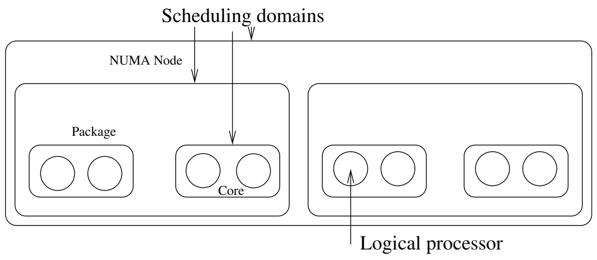


Figure 8: Scheduler domains in Linux. Each box represents one scheduler domain.

ality. Only the baseline implementation is included in the initial loadable kernel image, targeting the ISA supported by all the processors in a family. For example, the floating point state saving and restoring code will target the ISA supported by the Pentium processor in Intel® Architecture. The alternate implementations are stored in a special ELF section. After discovering the platform at boot time, the kernel walks through this special ELF section and the alternative implementations are patched into the kernel image. After the patching process is finished, the running kernel image behaves as it it has been configured/compiled for the exact platform it is running on. Examples of such patching in the Linux kernel include memory barriers, saving and restoring the floating point state, SMP primitives, and prefetch hints.

In the context of heterogeneous systems, this runtime patching mechanism may not work. The above described method of patching works because the alternatives mechanism assumes that the platform is homogeneous. This is an example of the optimizations that need to be redesigned in the context of heterogeneity. One crude method that we have implemented is by replacing the alternatives mechanism with real alternative code paths in the kernel image that check for the presence of the specific features. However, other optimizations in this area are possible.

The run–time code patching feature is specific to Linux and the BSD family of operating systems do not seem to have a similar feature.

5.3 System topology

Load balancing in multiprocessor systems depend on the accurate discovery of the system topology. In Linux, the complete system topology is divided into an elegant tree of scheduling domains which may share various parts of the architecture to a smaller or larger degree. An example of a scheduling domain is a group of cores that share the last level cache. An illustration of these domains can be seen in Figure 8. The load balancing code starts at the lowest level in which each domain is a logical processor and then moves up the system topology identifying imbalances in each domain. If it finds an imbalance, it attempts to rectify it. This algorithm assumes that the topology is balanced or symmetric. However, in case of asymmetric core architectures, there is no guarantee that the system topology will be balanced.

One illustration of a scheduling domain asymmetric topology is shown in Figure 9. In such a topology, as we reach the system topology nodes which are not balanced themselves,

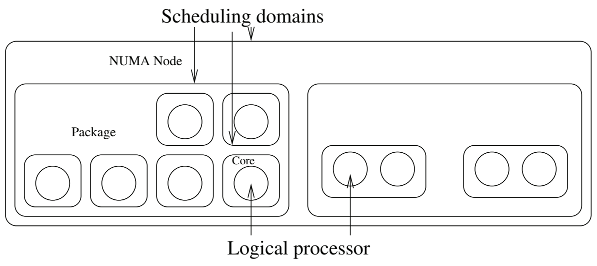


Figure 9: Scheduler domains in asymmetric topologies. Load balancing needs to account for differences in topology and compute power.

the balancing acts at the *Numa node* domain will result in an imbalanced system, because the child domains, which are the *Packages* have different numbers of logical processors in them. One way to mitigate this is to quantitatively take into account the *compute power* of a scheduler domain itself. This will allow us to balance across asymmetric topologies because their asymmetries will be masked by the *compute power* factor.

5.4 System timekeeping

Linux uses the timestamp counter (TSC) for fine–grained time keeping. Coarse–grained time keeping can be done using a variety of timer options available on the platform. However, the TSC runs at the frequency of the processor clock. Linux calibrates the TSC using a known external timer once on the BSP and assumes that it is the same for all the APs. In a heterogeneous systems it is easy to see that we may have cores that run at different frequencies. This introduces a small error every time we do the standard time measurement which then builds cumulatively. Over time (which can be fairly small in wall clock terms) it can become a source of significant error in idle events and fine grained timing related tasks such as those in the file system which depend on *gtod*. As discussed in Section 3.6, the lack of a fixed rate TSC in the E5450/N330 prototype leads to unreliable timing, so our system uses a core-independent timer source (the *HPET*) for timing purposes.

5.5 Trampoline code

The bootstrap process in a typical multiprocessor requires that the BSP send an inter–processor interrupt to each AP along with a vector. The vector address is the address of the trampoline code. The Linux kernel assumes that this trampoline code is the same for all the processors, which could break in the presence of heterogeneity. In the current processor architectures, the trampoline code more or less handles all the processors in a single processor family. However, it is conceivable that if there are cores with different microarchitecture and feature sets that differ slightly in their initial bootstrap process, the bootstrap code need to be custom for the different processors. This aspect of heterogeneity did not have an impact in any of our prototypes.

5.6 Power efficient idling

Since the processor cores can be in an idle state for significant amount of time, idle state management is a fairly important job of the Linux kernel. The *cpuidle* driver handles the idle–state management in the Linux kernel. The device driver developers can provide hints to the idle management subsystem about the tolerable latencies in a given device state. The *cpuidle* driver ensures that there is only one idle loop algorithm for all the cores in the system. However, each of the cores can be in a different state at any given point of time. The idle states are initialized only once for the BSP and it is assumed that the same hold true for all the cores. On one of our hardware prototypes the Linux kernel used only the C1 state idle handling using the *MWAIT* instruction. Ideally, one would expect that the operating system would use per–cpu idle loops that are optimized for a given core architecture for maximal power savings and appropriate entry and exit latencies. For a heterogeneous systems, Linux should have a cpuidle driver per–cpu which could be different and uses the C–states that are specific to the core.

5.7 Dynamic voltage and frequency scaling

Modern processors support a wide range of operating system visible dynamic voltage and frequency scaling (DVFS) states. DVFS allows for a nice method to save battery power as lower clock speeds (at low voltages) mean lower power consumption. One can think of various governors that can act on different cores at different times in heterogeneous systems. As in the case of the *cpuidle* behavior, the Linux kernel has a single driver that provides the mechanism for driving the various cores in the platform to different P–states (The ACPI term for various performance/DVFS states). During initialization, this driver populates the various P–states that the core that it ran on is capable of transitioning to. This information is copied into all the per–cpu information nodes. On a heterogeneous system this is not a correct assumption because all the cores may have different sets of possible P-states. Thus, we need a per core type mechanism (or more generally a per–core) mechanism to drive the individual cores to different P–states as is appropriately chosen by a myriad of governing policies.

5.8 Machine check architecture

Intel® Architecture processors implement a mechanism that can detect and report hardware errors called the Machine Check Architecture (MCA). To this extent they consist of several error–reporting *register banks*. Each bank is associated with one or more hardware units. Our E5450/N330 hardware prototype consisted of cores that supported different numbers of MCA banks. The Linux kernel assumes that the banks discovered in the BSP are applicable to all the APs. In fact, it forces the number of banks to be the same. The actual meaning of the error codes reported is likely to be different for different cores in a heterogeneous system. Thus, the identification of the MCA and the subsequent hooking up of the error decoding notifier chain should be made core type specific in heterogeneous systems.

5.9 Non–architectural features

The Linux kernel sometimes uses the non–architectural features such as model–specific registers (MSR). While use of such features in a homogeneous system will not create any problems, in case of heterogeneous systems, such features always need to be conditional on the individual core type. One example of such a use is the last branch record (LBR) feature in the *perf* subsystem. When these registers need to be reset, the addresses of these registers are model–specific, and differ between the E5450 and N330 processors. The Linux kernel, assumes that the registers are same, thus resulting in machines crashing in strange ways.

Operating systems have generalized certain properties of the cores (such as the C–states as well as P–states) as platform devices. Since, in the case of homogeneous systems the mechanism and the possible states of the cores are identical, this approximation holds. However, in case of heterogeneous systems, the possible states as well as the mechanism to change state differs, these devices need to be per–core with distinct properties in terms of states as well as mechanisms. In many cases, the assumption on homogeneity is built-in in the many data structures the kernel handles. While we cover some specific examples in the previous sections, more generically the kernel needs to assure that any data structure that is CPU specific is made per-CPU. For example, in the case of our E5450/N330 system, the N330 processor features simultaneous multithreading while the E5450 does not. In Linux, a single variable accounts for the number of hardware threads that a single core supports, leading to incorrect construction of the scheduler domains hierarchy in this system.

6. RELATED WORK

Prior work has demonstrated the benefits of single ISA heterogeneous architectures in achieving improved performance per watt. Kumar et al. [15, 16] demonstrated the performance benefits of single ISA heterogeneous architectures. Annavaram et al. [1] varied the amount of energy expended to process instructions according to the amount of available parallelism. Ghiasi et al. [7] demonstrated improved efficiency by using application characteristics to control dynamic voltage and frequency scaling. Hill and Marty [10] show that asymmetric multi-core designs provide greater potential speedup than symmetric designs.

Knauerhase et al. [13] demonstrated the benefits of dynamic runtime observation in the operating system to optimize performance on heterogeneous architectures. Koufaty et al. [14] added bias scheduling for heterogeneous systems to match threads to the core type that can maximize system throughput, based on dynamically matching workload characteristics to core microarchitecture. Shelepov and Fedorova [28] proposed Heterogeneity-Aware Signature-Supported scheduling using per-thread architectural signatures to avoid the need for dynamic profiling.

The majority of prior research assumes either single or disjoint ISAs. Our work lies in between and considers overlapping ISAs, which we believe is more practical. For disjoint ISAs, most manage the "different" cores as coprocessors or peripherals and incur high overhead when moving contexts across address spaces. CUDA® exposes graphics processors as a coprocessor through libraries and OS drivers [22]. Cell® offloads predefined code blocks to Synergistic Processor Elements [8] and EXOCHI offloads to a graphics processor via libraries and compiler extensions [29]. These designs place great burden on programmers, whereas we allow the operating system to transparently manage all cores as traditional CPUs.

Li et al. [19] presented a comprehensive study of OS support for heterogeneous architectures in which cores have asymmetric performance and overlapping, but non-identical instruction sets, enabling transparent execution and fair sharing of different types of cores. We complement this work by presenting the detailed analysis of a real heterogeneous platform.

MISP [9, 29] employs proxy execution similar to fault-and-migrate. However, it requires hardware support for user-level faults and inter-processor communication. Hypervisors such as Xen [3] and VMware [25] use fault and emulate techniques to implement instructions not supported by hardware vitalization features.

Existing operating systems which expose heterogeneity generally avoid resource management issues by having secondary schedulers for the heterogeneous resources, accessed via a device driver model. Probably the most prevalent example of this is the usage of device drivers for graphics cards which hide domain specific compute engines.

In the research community, Helios [21] introduces satellite kernels to export a single, uniform set of OS abstractions across heterogeneous systems, retargeting applications to available ISAs by compiling applications to an intermediate language. Barrelfish [4] takes a distributed system approach to hide CPU asymmetry behind 'cpu drivers', exposing CPU capabilities directly to applications which must adapt based upon the dynamic view of the system held in a system knowledge base.

7. CONCLUSIONS

In this paper, we analyze various methods that can be used to bridge the functional issues that arise in a heterogeneous systems which has processor cores from different microarchitecture families. We explore the design choices that are available for software designers and discuss the pros and cons of each. We find that the software stack including operating systems and applications will need to adapt to exploit the heterogeneous nature of the underlying platform. We have prototyped some of our software modifications in two hardware prototypes of heterogeneous systems. We conclude that such modifications will provide for the smooth adoption of heterogeneous systems into the computing realm in a backward compatible manner. Even as we enumerate the software choices to overcome heterogeneity, there is no once choice that suits all software stacks. The design goals of individual software stacks will certainly influence the way in which they approach heterogeneity. Although our study and experience have a number of limitations, our findings can influence the direction in which operating system evolve as well as hardware design choices. By way of hardware and software prototypes, we identify several assumptions about homogeneity that the current operating systems make. Further, our experience with the hardware prototyping emphasizes that a symbiotic evolution of hardware and software is necessary for this category of heterogeneous systems to fulfill its potential.

8. REFERENCES

[1] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's law through EPI throttling. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 298–309, June 2005.

[2] A. Barak and O. La'adan. The mosix multicomputer operating system for high performance computing. In *Future Generation Computer Systems*, 1998.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, New York, NY, USA, Oct. 2003. ACM.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schűpbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, pages 29–44, New York, NY, USA, Oct. 2009. ACM.

[5] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, Apr. 2005.

[6] F. A. Bower, D. J. Sorin, and L. P. Cox. The impact of dynamically heterogeneous multicore processors on thread scheduling. *IEEE Micro*, 28(3):17–25, May/Jun 2008.

[7] S. Ghiasi, T. Keller, and F. Rawson. Scheduling for heterogeneous processors in server systems. In *Proceedings of the 2nd Conference on Computing Frontiers*, pages 199–210, May 2005.

[8] M. Gschwind. The Cell* broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35(3), June 2007.

[9] R. A. Hankins, G. N. Chinya, J. D. Collins, P. H. Wang, R. Rakvic, H. Wang, and J. P. Shen. Multiple instruction stream processor. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 114–127, June 2006.

[10] M. Hill and M. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, July 2008.

[11] Intel Corporation. *Intel$^{®}$ 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference*. Intel Corporation, June 2009.

[12] Intel Corporation. Intel$^{®}$ Virtualization Technology FlexMigration Application Note 323850. http://www.intel.com/Assets/PDF/manual/323850.pdf, May 2010.

[13] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multi-core systems. *IEEE Micro*, 28(3):54–66, May 2008.

[14] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the Fifth European conference on Computer Systems*, New York, NY, USA, Apr. 2010. ACM.

[15] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, Dec. 2003.

[16] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 64–75, June 2004.

[17] T. Li, D. Baumberger, D. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, Nov. 2007.

[18] T. Li, P. Brett, B. Hohlt, R. Knauerhase, S. D. McElderry, and S. Hahn. Operating system support for shared-isa asymmetric multi-core architectures. In *Proceedings of the Fifth Annual Workshop on the Interaction between Operating Systems and Computer Architecture*, June 2009.

[19] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-isa heterogeneous multi-core architectures. In *Proceedings of the Sixteenth International Symposium on High-Performance Computer Architecture*, Jan. 2010.

[20] C. Morin, R. Lottiaux, G. VallÃl'e, P. Gallard, G. Utard, R. Badrinath, and L. Rilling. Kerrighed: a single system image cluster operat-ing system for high performance computing. In *Proceedings of the 9th International Euro-Par Conference*, Aug. 2003.

[21] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, pages 221–234, New York, NY, USA, Oct. 2009. ACM.

[22] NVIDIA. *NVIDIA CUDA Programming Guide, Version 1.1*. NVIDIA Corporation, Nov. 2007.

[23] S. Parekh, S. Eggers, H. Levy, and J. Lo. Thread-sensitive schedling for smt processors. 2000.

[24] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first generation CELL processor. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 184–185, Feb. 2005.

[25] M. Rosenblum and T. Garfinkel. Virtual machine monitors: Current technology and future trends. In *Computer*, volume 38, pages 39–47, May 2005.

[26] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the Fifth European conference on Computer Systems*, pages 139–152, New York, NY, USA, Apr. 2010. ACM.

[27] S. Saisanthosh Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 506–517, June 2005.

[28] D. Shelepov and A. Fedorova. Scheduling on heterogeneous multicore processors using architectural signatures. In *Proceedings of the Fourth Annual Workshop on the Interaction between Operating Systems and Computer Architecture*, June 2008.

[29] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. EXOCHI: Architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 156–166, June 2007.