Kinship: Efficient Resource Management for Performance and Functionally Asymmetric Platforms*

Vishakha Gupta
Intel Labs, Hillsboro, OR, USA
vishakha.s.gupta@intel.com

Rob Knauerhase
Intel Labs, Hillsboro, OR, USA
knauer@jf.intel.com

Paul Brett
Intel Labs, Hillsboro, OR, USA
paul.brett@intel.com

Karsten Schwan
Georgia Institute of Technology, Atlanta, GA, USA
schwan@cc.gatech.edu

ABSTRACT

On-chip heterogeneity has become key to balancing performance and power constraints, resulting in disparate (functionally overlapping but not equivalent) cores on a single die. Requiring developers to deal with such heterogeneity can impede adoption through increased programming effort and result in cross-platform incompatibility. We propose that systems software must evolve to dynamically accommodate heterogeneity and to automatically choose task-to-resource mappings to best use these features.

We describe the *kinship* approach for mapping workloads to heterogeneous cores. A hypervisor-level realization of the approach on a variety of experimental heterogeneous platforms demonstrates the general applicability and utility of kinship-based scheduling, matching dynamic workloads to available resources as well as scaling with the number of processes and with different types/configurations of compute resources. Performance advantages of kinship based scheduling are evident for runs across multiple generations of heterogeneous platforms.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design, Performance

Keywords

Performance asymmetry, Functional asymmetry, Kinship, Dynamic scheduling

*This research was funded in part by the National Science Foundation SHF and Intel Corporation.

CF'13, May 14–16, 2013, Ischia, Italy.

1. INTRODUCTION

Heterogeneity in hardware. Leading architects have observed that microprocessor performance has grown 1,000-fold over the past 20 years, driven by transistor speed and energy scaling, as well as by micro-architecture advances that exploit density gains from Moore's Law [2]. However, diminishing transistor-speed scaling and practical energy limits are creating new challenges for continued performance scaling. This has forced processor designers to use large-scale parallelism and explore heterogeneity, e.g. accelerators, to achieve performance and energy efficiency. The resulting *rapid evolution in functionally and performance asymmetric platforms* is evident from recent industry efforts like Intel's QuickIA evaluation system [3] and AMD's Fusion architecture.

Diversity in applications. Large data center and cloud installations now provide high end computing to common users in the forms of gaming clouds and high quality media delivery and processing (e.g., Netflix streaming), and to corporate customers running engineering simulations or financial codes [17]. IT providers support this range of high end applications with data center systems that employ functionally asymmetric platforms like GPU-based accelerators [6]. These systems, then, run dynamic mixes of applications that create *a time-varying diversity of workloads* for data center platforms. This is the case for both consolidated (virtualized) and non-consolidated systems, the latter because most realistic applications have complex behaviors with multiple phases or stages that differ in their relative compute, IO, or memory intensities.

Implications on systems software. The aforementioned trends in hardware evolution and application diversity present challenges, particularly when continuing along the current path of development in which few applications are customized to take advantage of asymmetric hardware. For instance, CUDA-like programming models address only certain accelerators, compilers' abilities to deal with asymmetries are limited when hardware evolution is uncertain and/or when it is shared by multiple applications. *It is imperative, therefore, for the systems software to offer solutions that address the dynamic range of future platform asymmetries.*

Kinship. This paper presents a complete systems solution and associated theoretical framework based on a unifying *kinship metric*. Jointly, these provide support for asymmetry-aware operation on future multi–core platforms. At the systems software level, Kinship is used to schedule diverse *schedulable entities* such as operating system threads or virtual machines' (VMs') virtual CPUs (VCPUs), onto rapidly evolving asymmetric hardware. Termed *VCPUs* for generality in the remainder of this paper, kinship-based scheduling for such entities uses: (1) on–demand VCPU migration (fault-and-migrate) to cope with functional asymmetry, (2) online

monitoring and hint mechanisms to determine VCPU diversities, and (3) leverages these capabilities for effective runtime VCPU-PCPU(physical CPU) mappings.

We experimentally demonstrate Kinship to be a useful runtime abstraction for heterogeneous hardware and make the following contributions towards system software support for heterogeneity:
(1) A rigorous theoretical formulation defining the kinship metric.
(2) Asymmetry-aware scheduling based on the kinship metric, that can effectively match dynamic workloads to diverse resources; this improves on previous work limited to specific asymmetries or simply sharing the most powerful cores equally among all workloads.
(3) Additional resource monitoring and management that can cope with the evolution of hardware asymmetries; the claim is demonstrated with notable performance gains on three different asymmetric hardware platforms with an implementation of kinship scheduling in the Xen hypervisor.

The Montage prototype. The Montage prototype implements Kinship and associated methods as an extension of the Xen hypervisor, thereby creating asymmetry-capable virtual platforms able to run arbitrary systems and applications. For the asymmetric platforms it targets, Montage offers substantial performance gains compared to existing asymmetry-unaware scheduling methods like the current Xen credit scheduler. For instance, with functional asymmetry due to the lack/presence of cryptographic instructions, 2X performance improvements are seen on Xeon platforms (see Section 5), with negligible standard deviation in per-run performance.

2. MACHINE MODEL

We briefly outline the asymmetries that Kinship addresses:

Performance asymmetry: covers variations in processing capabilities due to core frequency scaling, in-order vs. out-of-order behavior, internal instruction issue/commit widths, last-level cache sizes, etc. Irrespective of the cause of performance asymmetry, the outcome is the difference in performance observed for application processes running on different cores.

Functional asymmetry: reflects the importance of accelerators, where some cores on the chip are better customized for certain tasks than others. Currently, we assume the existence of a common subset of ISA on all cores, with some cores enhanced by special instructions to support certain classes of applications. The fault-and-migrate model permits applications to run on such 'shared ISA' cores [15], where a VM's VCPU experiences a fault when it uses an 'unsupported' instruction, causing the hypervisor to re-map the VCPU to a PCPU supporting the instruction and then continue its execution. Although CPUID (CPU identification instruction on x86) could be used to predict faults in advance, such static provisioning will reduce platform utilization, particularly when faulting instructions are rarely invoked.

Intel's experimental *QuickIA* platform and ARM's big.LITTLE are current examples of such designs. With functional asymmetry, the challenge is to correctly associate VCPUs with PCPUs, so as to minimize the risk of faulting. Dealing with performance asymmetry requires effective VCPU-to-PCPU matching; our kinship approach, dealing with both, is described in the next section.

3. THE KINSHIP MODEL

To address the kinds of asymmetries described above, *kinship-based scheduling* makes decisions based on *kinship values* between the VCPUs of guest domains and the PCPUs present in the system.

Kinship or the kinship metric is "a numerical value computed based on both workload and physical hardware characteristics, reflecting the best current match of VCPUs to PCPUs, to enable efficient execution and effective utilization". We formulate Kinship such that its parameters are (1) measurable and (2) have a quantifiable effect on performance of a schedulable entity, i.e., a VCPU. The parameters used to calculate kinship values are calibrated for the system and/or supplied by an external source, or they are populated by the system at runtime. Without external input, the model starts with values that represent the most common behavior; they are updated as VM execution progresses.

3.1 Generic Resource Representation

We characterize the resources that cause asymmetries and play a role in determining the scheduling of VCPUs on corresponding PCPUs as follows. Let R be the set of all resources in the system. Let R_x, R_y be subsets of R. Some R_x and R_y may have shared resources e.g. $\{cache, memory, IO, \dots\}$ or they may have some exclusive resources like $\{cpu\}$, and $R_x \cap R_y \neq \emptyset$, in general. This set is determined by the measurable elements of the system. Since our objective is to find optimal mappings of VCPUs to PCPUs, the definition of kinship associates all of these resources with individual PCPUs in the system. In other words, R_x in our discussion is the set of resources associated with some PCPU, p. As stated earlier, different PCPUs can have common resources, in one socket sharing a cache, or in current systems, all PCPUs share the memory and IO subsystems. For simplicity of explanation, assume the following set R_x of resources per PCPU $\{cpu_x, cache_x, mem_x, io_x\}$, where *cpu* is the processing core, *cache* is the last level cache in the socket, *mem* is the RAM accessible from the PCPU and *io* denotes the IO subsystem. It is of course possible to define as many elements in this set depending on the degrees of asymmetry, like L1 cache and individual processing units.

3.2 Kinship Formulation

Adopting a top-down approach for presenting the kinship model, Equation (1) shows the primary equation that evaluates kinship values per VCPU v and PCPU p pair.

$$K_p^v = E_p^v + F_p^v + C_p^v \tag{1}$$

where E_p^v and F_p^v are the performance and functional components of kinship K_p^v and C_p^v indicates whether v is allowed to run on p. All of the quantities discussed here are defined per VCPU-PCPU pair. Therefore, unless noted otherwise, we will omit the v and p indexing in the subsequent equations.

Asymmetries are classified based on performance vs. functional because of differences in the way these asymmetries manifest themselves and the effect they have on workloads. For instance, functional asymmetry like the absence of some group of instructions would cause a VCPU to fault and disrupt its execution, where-as a frequency difference between cores would simply result in different execution times witnessed by the workload on the different PCPUs. The model we describe incorporates such disparities. An additional important factor, especially in future large scale many-core systems, is the ability to limit the group of PCPUs on which a VCPU can be scheduled. This could be done for various reasons like memory distances, non-uniform cache effects, latency of communication between various resources, etc. C^v denotes such a permissible set of PCPUs.

Now, each component in Equation 1, at any time t, can be defined individually, and we can express K as a dot product of the weight vector with the vector composed of individual kinship components as shown in Equation 2.

$$\begin{aligned} &E = w_E * E_t;\ F = w_F * F_t;\ C = 0 \text{ if } p \in cpupool^v, \text{ else } -\infty \\ &K_p^v = \begin{bmatrix} w_E & w_F & 1 \end{bmatrix} . \begin{bmatrix} E_t & F_t & C_t \end{bmatrix}^\intercal \end{aligned} \tag{2}$$

The weights w_E and w_F control the effect of each performance and functional asymmetry component on the overall kinship value.

3.3 Performance Kinship

Performance kinship captures those elements in the system that can affect the performance achieved by individual applications and the overall system, e.g. diversity in applications or heterogeneity in hardware. We first show how kinship takes into account these various elements of performance asymmetry. Since kinship is expected to incorporate observed workload behavior, load on the resource under consideration at any given time, and potentially user-specified information, we define the following matrices of parameters addressing only the resource types being accounted for in the system. Let r denote a resource in our resource set R, discussed in Section 3.1. We henceforth use it to denote some resource belonging to a particular PCPU's resource set $r \in R_x$. We abbreviate $c = cpu$, $l = cache$, $m = mem$, $i = io$ due to space constraint in Equation (3), and use ρ to symbolize correlation between pairs of resources.

$$I_r^v = \begin{bmatrix} i_c & \rho_{c,l} & \rho_{c,m} & \rho_{c,i} \\ \rho_{l,c} & i_l & \rho_{l,m} & \rho_{l,i} \\ \rho_{m,c} & \rho_{m,l} & i_m & \rho_{m,i} \\ \rho_{i,c} & \rho_{i,l} & \rho_{i,m} & i_i \end{bmatrix}; L_r = \begin{bmatrix} u_c & \rho_{c,l} & \rho_{c,m} & \rho_{c,i} \\ \rho_{l,c} & u_l & \rho_{l,m} & \rho_{l,i} \\ \rho_{m,c} & \rho_{m,l} & u_m & \rho_{m,i} \\ \rho_{i,c} & \rho_{i,l} & \rho_{i,m} & u_i \end{bmatrix}$$

$$G_r^{v\intercal} = \begin{bmatrix} g_c & g_l & g_m & g_i \end{bmatrix}$$

$$W = \begin{bmatrix} w_c & w_l & w_m & w_i \end{bmatrix} \ldots \text{relative importance of resources} \tag{3}$$

In Equation (3), I_r^v is a matrix representing the intensity at which each v uses each r; L_r is a matrix representing the (cumulative) load currently experienced by r due to all other VCPUs scheduled to use it; Since any VCPU v's use of one type of resource r_m (e.g. memory) associated with PCPU p can possibly have an effect on its use of another resource r_n (e.g. cache) associated with p, we use matrices with ρ representing this correlation in case of I and L. However, to reduce the extreme complexity in this case, we ignore these second order effects and simplify equations for I_r^v and L_r by equating the ρ values to 0. For instance, consider I_r^v after simplification. When a memory intensive VCPU runs on some PCPU, we observe performance counters and conclude that the VCPU is memory intensive, shown by the value i_m. However, that observation will be independent of the observation that will tell us about its cache utilization, captured in i_l. The remaining vector from Equation (3), G_r^v is a vector of values computed from given hints about v's usage of r normalized to the minimum capability of all resources of the same type in the system. G_r^v is simply a vector instead of a matrix because a user specifies per resource expectations or hints for each VCPU for now (future work will look at how user could specify other hints like coördinated execution).

Now, for a given $\{p, v\}$ pair, we define performance kinship, at any time t, as shown in Equation (4).

$$E_t = W.I^v.L.G^v \tag{4a}$$

$$\text{which computes to} \quad E_t = \sum^{r \in R_x} w_r * I_r * L_r * G_r$$

$$\ldots \text{where } R_x \text{for p} = \{cpu, cache, mem, io\} \text{ in our equations}$$

$$\ldots \text{OR can be expressed as}$$

$$E_t = w_{cpu} * E_{cpu} + w_{cache} * E_{cache} + w_{mem} * E_{mem} + w_{io} * E_{io} \tag{4b}$$

The individual elements from Equation (4).b are calculated as shown in Equations (5), (6), (7), (8) and explained in the remainder of this section. We use $p.readyQ$ to represent all VCPUs assigned to p in these equations. In the equations above, L_{cpu} is the *strength coefficient* or *load factor* for the PCPU. Its computation iterates over all VCPUs assigned to the PCPU at the given time and accounts for the load they have introduced on this PCPU, in order to give the remaining strength or capacity available for the VCPU being considered. Thus, $IOload_p$ accounts for time p would spend waiting on IO for v, hence increasing the strength value and therefore the Kinship. $CPUload_p$, on the other hand, lowers the kinship, as desired.

$$E_{cpu} = I_{cpu} * L_{cpu} * G_{cpu} \quad \ldots \text{For CPU}$$

$$I_{cpu} = \text{observed CPU intensity from performance counters}$$

$$L_{cpu} = \begin{cases} 1 & \text{at no load,} \\ \frac{1+IOload_p}{CPUload_p} & \text{otherwise} \end{cases}$$

$$IOload_p = \sum^{p.readyQ} \frac{I_{io}^v * CC^v}{CC_p^{min}}$$

$$CPUload_p = \sum^{p.readyQ} \frac{I_{cpu}^v * CC^v * CS^v}{CC_p^{min}}$$

$$G_{cpu} = \frac{S_p * EXP_{cpu}^v}{S_{min}} * S_p$$

$$S_p = \text{Calibrated speed for } p;\ S_{min} = \text{Slowest speed } \forall p \tag{5}$$

The rate at which a VCPU executes in the system usually depends on the relative speed of the cpu core it runs on. If an execution expects a certain level of performance, say, conveyed by a QoS agreement, core asymmetry cannot be ignored. For credit-based scheduling, this means that we must adjust the credits that specify the proportion of CPU time received by the schedulable entity: we define $CC^v = \frac{S_{min} * credits^v}{S_p}$, called VCPU v's current credit value, and use it in calculating $CPUload_p$ and $IOload_p$. We scale the credits available to a VCPU based on the speed (S_p) of the PCPU p under consideration, and the minimum across all PCPUs in the system. For example, on a PCPU $p2$ with a speed of 3GHz and a minimum PCPU speed in the system of 2GHz, a VCPU with credits of say, 120 will receive a CC value of 80 on $p2$. This accounts for the differences in speeds of the various PCPUs and leads to fairness [14] in PCPU cycles assigned to the VCPUs in the system. Thus, the expectations (EXP) and credit values are all defined with the slowest CPUs or smallest cache sizes as the base. CC^{min}, the minimum CC value across all VCPUs, normalizes the load values with respect to the lowest seen, giving a proportional boost to VCPUs that have higher QoS expectations. The CS^v value used in calculating $CPUload_p$ incorporates the current runstate of a VCPU v and is defined as $EXP_{cpu}^v * runstate^v$, where $runstate^v$ is 0 if v is blocked (which implies that v is not using cycles on p) or 1 if running (which implies that v is using p cycles proportional to its expectation EXP). This can also be used to address P-state changes (platform power scaling).

Lastly, in Equation (5), G_{cpu} is the scaled expectation calculated by using the given expectation (EXP, discussed further in Section 4.2.2) for a VCPU, S_p and S_{min} similar to the scaling above. S_p value for all PCPUs and S_{min} are calibrated at runtime using Montage calibration code since Xen does not recognize asymmetries.

Similar to $r = cpu$ from Equation (5), in Equation (6), G_{cache} is the scaled expectation as calculated above by using the given expectation for a VCPU, the cache size (l) of the PCPU p under consideration, and the minimum across all PCPUs in the system. The l_p values for all PCPUs is acquired from the Montage calibra-

tion code. The runstate considered for cpu does not affect the cache footprint in an obvious way and hence, *CACHEload* just accounts for the expected cache usage, if specified, and the observed cache usage intensity [8].

$$
\begin{aligned}
&E_{cache} = I_{cache} * L_{cache} * G_{cache} \quad \ldots \text{For cache} \\
&\quad I_{cache} = \text{observed cache intensity from performance counters} \\
&L_{cache} = \begin{cases} 1 & \text{at no load,} \\ \frac{1}{CACHEload_p} & \text{otherwise} \end{cases} \\
&\quad CACHEload_p = \sum^{p.readyQ} working\_set^v * EXP^v_{cache} \\
&G_{cache} = \frac{l_p * EXP^v_{cache}}{l_{min}} * l_p \\
&\quad l_p = \text{Calibrated cache for } p;\ l_{min} = \text{Smallest cache } \forall p
\end{aligned} \tag{6}
$$

Equations (7) and (8) are simplified compared with CPU and cache because precisely quantifying the different ways in which memory and IO affect workloads is beyond the scope of our current work. Therefore, we primarily rely on the given hints from a user, as of now, to determine the behavior of the VCPU showing memory or IO intensity. However, we can characterize a VCPU as IO intensive using performance counters like RESOURCE_STALLS and LLC_MISSES, described further in Section 4.2.3.

$$
\begin{aligned}
&E_{mem} = I_{mem} * L_{mem} * G_{mem} \quad \ldots \text{For mem subsystem} \\
&L_{mem} = \sum^{p.readyQ} EXP_{mem}; \quad G_{mem} = EXP_{mem}
\end{aligned} \tag{7}
$$

$$
\begin{aligned}
&E_{io} = I_{io} * L_{io} * G_{io} \quad \ldots \text{For the IO subsystem} \\
&L_{io} = \sum^{p.readyQ} EXP_{io}; \quad G_{io} = EXP_{io}
\end{aligned} \tag{8}
$$

The L_r values capture the current load on a resource. During the course of our evaluation, we realized that it was important to account for load introduced by VCPUs on all resources while calculating the kinship value of a new VCPU to the PCPU being considered. Therefore, we have now enhanced all the E_r computations, shown in Equations (5), (6), (7) and (8) to use $L_{overall}$ which is computed as $L^v_{overall} = \sum^{allvcpusonp} \left\{ \sum^{r \in R_x} L_r \right\}$. The L values are defined to become mathematically smaller as load increases (e.g., inversely proportional to the cpu/cache-intensity of VCPUs assigned to PCPU p), reducing the overall kinship values in Equation (4). Also, for the evaluation presented in Section 5, we provide EXP values for all VCPUs to indicate their resource expectations.

3.4 Functional Kinship

We define F_t from equation (2) as $F_t = MF * FV$. MF here denotes the match factor between a VCPU v's expected category (CAT^v) and a PCPU p's capabilities (CAP_p). The VCPU categories are discussed further in Section 4.2.2. PCPU capabilities are identified when their characteristics are being calibrated. For example, if 4 out of 8 CPUs in the platform support SSE extensions while also supporting all other general purpose instructions, those four can be marked $VECTOR$ along with $GENERAL$. So, different PCPUs can display multiple overlapping capabilities or have completely disjoint capabilities like that of a GPU (which would purely be $VECTOR$) and CPU (which would be $GENERAL$). *This will allow us to accommodate disjoint ISA CPUs in the future*. In our implementation, these capabilities are captured as bitmaps that can be *AND*ed for quick calculation. Hence, in Equation (9a), $match = CAT^v \& CAP_p$.

$$
MF = \begin{cases} 1 & \text{if } match = 0, \\ match & \text{otherwise} \end{cases} \tag{9a}
$$

$$
FV = \begin{cases} 1 & \text{if } faults^{TW} = 0, \\ faults^{TW} * \left(-1 + \frac{emulated}{1+emucost} \right) & \text{otherwise} \end{cases} \tag{9b}
$$

FV is the fault value as calculated in Equation (9b). We add up the observed number of faults when a VCPU faults on a particular PCPU over a moving time window TW. It is possible for a fault to lead to an emulation code which would be slower but won't halt the VCPU execution or cause migration. That is taken care of in the second term of Equation (9b). The variable *emulated* is a boolean value equal to either 0 or 1 depending on presence or absence of emulation. If *emulated* is 1, the negative effect of faults is reduced by its presence in inverse proportion to the cost of emulation (or the slowdown observed).

3.5 Kinship in Practice

To explain kinship practically, we construct challenging usecases with representative workloads from the Parsec suite, Hadoop sort [sort-1G] and Hadoop wordcount [wc-240M], Iozone, and an Intel-published encryption benchmark (AES-bench).Use cases are purposely defined in ways that do not make it easy for kinship scheduling due to mixing of computationally demanding vs. I/O intensive codes, and codes suitable for "small" cores vs. those requiring "big" cores, at an application level.

***Use-case1* for performance asymmetry**: consider scheduling two sets of applications: one set, denoted VM1, runs *ferret*, *Iozone* for a file size of 512MB in automatic mode; the other set, denoted VM2, runs *streamcluster (sc)*, *freqmine*, *hadoop-sort* of GB-size data and another similar *Iozone* instance. Of these benchmarks, Iozone, freqmine, and sorting are mostly unaffected by the speed and cache size of the Atom processor (see the performance comparison in Section 5). In this use case, kinship scheduling must answer the question "How do you schedule these application sets on a dual-socket *QuickIA* platform composed of a quad-core Xeon socket and a dual-core, hyper-threaded Atom socket?" Intuitively, the computationally demanding benchmarks in these sets should run on the Xeon, whereas the codes that are I/O-intensive should use the Atom. Kinship scheduling should act accordingly.

***Use-case2* functional asymmetry:** certain compute-intensive VCPUs may run better on slower cores when those have certain heavily used special instructions. Consider VM1 running *AES-bench* with a large data set and *dedup* on its two VCPUs, and VM2 running *AES-bench* with a small data set and *swaptions* on its two VCPUs. *Dedup* and *AES-bench-small* are marginally affected by speed, whereas *swaptions* and *AES-bench-large* suffer significant slowdown on slow cores. AES-bench benefits from AESNI instructions, so is greatly slowed down if AES is emulated in software. Experimental evaluations with these two VMs use an asymmetric platform with four Xeon cores across two sockets (2 cores from Socket1 and two from Socket2), where the AESNI instruction is disabled in Socket1, forcing software emulation. "Small cores" are created by slowing down one core from each socket by 50%. The challenge for kinship scheduling is "how to utilize fast instructions available only on certain cores?"

Given these parameters for E^v_p, initially, there is no observed behavior for Use-case1 and hence, E^v_p is not affected by it for any of the VCPUs. Assuming equal credits for both VMs, the difference will be visible among the VCPUs with regard to their CPU/memory expectations. VCPUs running ferret and streamcluster could have hints labeling them as MOSTLY_CPU. This will lead to a higher

value of kinship between those VCPUs and the PCPUs with better computational capabilities (i.e., the Xeon cores). The rest of the VCPUs running freqmine, Iozone, and hadoop-sort will get a chance to use the Xeons only when those cores are not currently used. Else, these workloads will be scheduled on the Atoms.

Considering Use-case2, the two VCPUs running AES-bench will belong to the same category, $CRYPTO$. The Xeon cores being the targets in this scenario are marked as $CRYPTO$ capable if they support the AESNI instructions. Hence, compared to the other VCPUs, the CRYPTO VCPUs will have a higher functional kinship value for the corresponding CRYPTO PCPUs. Kinship-based scheduling, however, considers both functional and performance asymmetries. As a result, since the particular platform under consideration also has speed asymmetry, the performance component of kinship will result in the VCPU running the large AES data set to have the highest kinship with the fast core supporting AES, whereas the smaller AES is scheduled on a core with AES support but only half the performance. Further, since swaptions is more compute intensive than dedup, it will have higher kinship with the faster core. Results of this execution are shown in Section 5.

3.6 Boundedness Analysis

In a typical implementation, the operations requiring the computation of kinship values are (1) adding or deleting an entity, which can change the kinship-based ordering, (2) scheduling an entity each scheduling cycle, and (3) recalculating the kinship values and overall ordering when observed parameters change their values or when users provide new hints. The common portion across all of these operations is the kinship calculation itself and a potential reordering from highest to lowest kinship values. The complexity of the actual calculation in Equation (1) is $O(r)$, where r is the number of resources, which in typical cases can be expanded as shown in Equation (4). Hence, it is a constant for all practically possible asymmetries. Scheduling and the periodic accounting require maintaining the scheduling order, based on the kinship value, so that the scheduler can assign PCPUs in that particular order, from the entity with highest kinship value to lowest. Using appropriate data structures, the cost of this operation can be lowered as the system reaches stable state. Its worst case complexity is $O(M*N)$, where M and N are the maximum number of PCPUs and VCPUs in the system. We evaluate the scalability of this re-calculation empirically through our Montage prototype in Section 5.6.

3.7 Summary

In summary, kinship is formulated to use both (1) user provided hints and/or dynamic runtime observations and (2) knowledge about platform configuration, resulting in the composite value K_p^v, used for asymmetric scheduling. The additive, multiplicative, and logical relations between the quantities are defined based on the effect a particular parameter can have if matched incorrectly to the system. E.g. CPU load is inversely proportional to the kinship value and can highly affect VCPU response time. Kinship factors this in by placing the load in the denominator.

We wanted kinship to be adaptable by various users. The relative "weights" in kinship account for other factors which are system dependent and can be adjusted by the user. E.g. for some systems where functional asymmetry is less important, the weight can be set to a low value. Figure 4 shows a sample of how functional weight can be effective. Since the VCPUs with highest kinship values are scheduled first, so as to allocate them to their best matched PCPUs, the kinship value K_p^v can be multiplied by a *fairness token*, which is a token passed around in a round robin order, to boost the holders priority for fast core access. This can help us distribute fast core cycles among all executables, similar to related work like [7]. Finally, we note that, while the text above focuses on hardware asymmetries for shared or single ISA architectures, the formulation can be extended to incorporate disjoint ISAs, by modest changes in the current equations.

4. MONTAGE

Montage is a software architecture for hardware platforms that display performance and functional asymmetries. It implements the kinship model described earlier to match schedulable entities to the physical platform based on their execution characteristics.

4.1 Montage Architecture

The software components of Montage are –

Asymmetry-aware hypervisor: is the core of Montage. Asymmetry awareness involves modifications to recognize asymmetric hardware and to schedule VCPUs via the kinship metric. Guest operating systems or applications need not be changed. VCPU behavior is observed using periodically inspected hardware performance counters (for E_p^v) and by monitoring instruction faults (for F_p^v) to help the hypervisor understand current VM behavior.

Cooperative platform interactions or hint channels: are for optimization, decoupling observation from scheduling and making it possible to treat scheduling and user-input problems separately. They permit guest VMs to inform the hypervisor about application characteristics/needs. This is useful because a guest is better positioned to understand the behavior of its applications, whereas the hypervisor has global knowledge about the hardware platform and the multiple VMs it is managing. In [5], we discuss ways to determine and then use appropriate hint channels.

Asymmetry-aware guest OS: can utilize asymmetries exposed and exploited by the asymmetry-aware hypervisor to also better schedule its tasks/threads. Such 'smart' guest-level scheduling can lead to additional performance gains when used with Montage.

4.2 Montage Implementation

The Xen credit scheduler schedules VCPUs from different guest VMs in proportion to their credits. Montage VCPU mappings are guided by the kinship model, but try to maintain the work conserving property of the Xen scheduler. For example, if PCPU $p2$ has an empty ready queue, and $p1$ has two VCPUs scheduled on it, the Xen scheduler will try to migrate one of the VCPUs to PCPU $p2$. This will be prevented in Montage if kinship values between the VCPUs and $p1$ are higher compared to their kinship values with $p2$. Instead, re-balancing in the next accounting cycle ensures a more even load distribution.

4.2.1 Kinship-based Scheduling

Kinship algorithms are implemented with optimized scheduling data structures and math operations as logical operators wherever possible, to reduce the overheads introduced by the runtime use of kinship equations, evaluated in Section 5.

Admitting a VCPU – In order to start a new VCPU on as well-matched a PCPU as possible, its kinship values with all PCPUs in the system are computed, incorporating all hints and capabilities. The VCPU is then placed in a kinship-ordered list of all VCPUs. Dynamic elements of resource loads are considered when the VCPU is actually assigned to a PCPU, as described next.

Assignment of a VCPU to a PCPU – The main scheduling loop considers the resource strengths (load factor), in the kinship value, in order to select the best PCPU at any given point in time. PCPUs are tagged for their speed and capabilities through runtime calibration. VCPU-PCPU placements are done periodically and automat-

ically, using dynamically updated kinship values that incorporate both hints and observations in the hypervisor scheduler. Hints play a larger role in the current version because a more sophisticated observation-based model involves significant complexity (e.g., detailed heuristics, potential inputs from compilers, etc.), but will be made optional due to our ongoing research on monitoring.

Accounting and re-scheduling – System configuration changes over time, e.g., new VCPUs are added, old ones removed, priorities change, etc. To maintain updated VCPU-PCPU mappings, a rematch algorithm is run every accounting period with a refresh of system state such as PCPU load information, to create new allocation opportunities for VCPUs. Re-matching is based on compilations of current and past observations, and on the initially provided hints. The accounting period for observations is chosen empirically so that such data changes gradually, the current period being four times the credit accounting period used by the Xen scheduler. Updates can be triggered sooner if frequent changes are made to VCPU or PCPU properties (e.g., software frequency scaling). VCPUs are migrated from the currently assigned PCPU to a new one, if necessary, and corresponding updates are made to the relevant data structures.

Additional modifications – More changes are required to make Xen asymmetry aware. With different generation cores in two sockets of a platform, the VM control structures (VMCS)—from processor VT extensions—have different formats, but the current Xen code maintains this information as globals assuming identical formats. We make these local to PCPUs to allow VMs to boot on asymmetric hardware. Calibration code calculates CPU speed differences, cache variations across sockets, and memory sizes accessible from each core. The calibration code times a loop executing different operations like addition, subtraction, division and multiplication on a set of variables, taking care to avoid compiler optimizations (necessary to get as accurate a calibration as possible). Calibration also checks for special instructions like the vector SSE and AESNI support, in order to correctly categorize PCPUs. Users are not involved in this.

4.2.2 Hint Channels

Using offline profiling or other information, users or administrators can specify the categories (CAT^v) and expected behavior (EXP) of the VCPUs belonging to some VM using the hypercalls described in [5]. The category values belong to a set {GENERAL, VECTOR, CRYPTO, ...}, where a VCPU can simultaneously belong to multiple categories depending on its workload. The default *CAT* for any VCPU is *GENERAL*. There are corresponding tags/capabilities (CAP_p) defined for the PCPUs as well, and they are used to calculate the match (MV) between a VCPU and a PCPU. Users can also specify expectations for VCPUs as {UNKNOWN, MOSTLY_CPU, MOSTLY_IO, ...}; they can be ORed together. The spin (loop) benchmark is an example of MOSTLY_CPU, while dedup is MOSTLY_IO. The default value is UNKNOWN.

4.2.3 Runtime Observations

The performance behavior of VCPUs is observed by counting events like OFFCORE_REQUESTS and RESOURCE_STALLS, which indicate a VCPU's bias for the faster or more complex core or larger cache (a similar approach is used in [9]). These counters are captured at each VCPU context switch, to attribute values to the appropriate VCPUs. For functional asymmetry, the Xen fault handler is changed to maintain the fault counts incurred by VCPUs. A fault lowers the kinship value between a VCPU-PCPU pair (as FV becomes more negative). An initial version of this observation algorithm monitors and maintains this information over a moving time window.

5. EXPERIMENTAL EVALUATION

Asymmetry provides opportunities to exploit workload diversity and/or dynamic workload change, to obtain improved hardware power/performance properties. Evaluation on realistic hardware demonstrates how kinship scheduling makes it possible to gain these advantages for evolving multicore platforms.

5.1 Benchmarks

For micro-benchmarks, we use *Spin*, an extremely predictable loop benchmark, and *CacheBuster*, which reads or writes elements of a byte array in a random order for a given number of iterations, using algorithms that stress the cache in various configurable ways. For functional asymmetry, we use AESBench, the Intel AESNI sample library implementation, which compares Gladman's AES performance with that of the AESNI optimized library. To better understand how different and dynamic performance behavior is captured by kinship, we use the SPEC CPU2006 suite, PARSEC suite, hadoop-sort, hadoop-wordcount, and Iozone benchmarks to create potential application scenarios. These represent the kernels of parallel codes running simulations along with the online I/O and analytics actions performed on simulation outputs, all running "in situ" on the same underlying hardware platform [22]. For standard servers running consolidated codes, they represent a combination of compute vs. I/O intensive applications.

5.2 Testbed and Methodology

The adaptable nature of kinship scheduling is demonstrated with machines exhibiting varying degrees of asymmetry.

SimulatedWM is a 12-core Intel® Xeon® 5645 processor-based machine with 12GB RAM and support for AESNI referenced in Use-case2 in Section 3. To evaluate kinship-based scheduling, functional asymmetry is obtained by disabling AESNI detection. Performance asymmetry is created using a combination of (1) duty cycle modulation (2) core frequency changes and (3) resizing the last level cache (LLC) using Intel-proprietary technologies. We create speed asymmetries at 100%, 75% and 50% of nominal frequency to create fast, medium, and slow cores, and cache asymmetries at 12MB and 3MB to create big and small caches.

QuickIA is an experimental 2-socket system containing a Intel® Xeon® X5450 with 6MB LLC and an Intel® Atom™ 330 with 512KB cache, sharing the main memory and other resources.

Dual Xeon is a prototype dual-socket platform, with socket0 consisting of four Intel® Xeon® E5440 cores @2.83GHz, 6MB LLC and socket1 with two Intel® Xeon® 5160 cores @3GHz, 4MB LLC. All cores share the 2GB RAM and remaining resources on the platform. This provides moderate asymmetries.

Each machine uses Xen-4.0 (testing) with Linux 2.6.32.41-pvops kernel as Dom0. Virtual machines run Fedora 12. Experimental scenarios are chosen to match workload mixes observed in cloud and data-center systems. For each workload, we present the average running time (over m iterations) of a VM running n consecutive workload instances, with floor and ceiling numbers to give the worst and best results as predicted by an oracle that is experimentally (but not provably) optimal. To remove the effects of the asymmetry unaware OS scheduler, we pin the benchmarks to equal number of guest VCPUs. We also set all weights in the equations to 1 for most of our experiments unless stated otherwise.

5.3 Scheduling for diversity

Workloads on the *QuickIA* platform show distinct differences in application performance (see Figure 1) depending on their behavior and the core mappings being used. Specifically, the performance of the computationally intensive PARSEC workloads is heavily af-

fected by core type for their native (large) data sets. In contrast, the memory/IO-bound nature of freqmine, sort, wordcount and Iozone results in modest performance differences on the platform's asymmetric processors. This combination of benchmarks also represents the different phases certain applications execute in datacenter systems in which, say, media transfer or caching is mixed with media processing [1], or on high end machines where computational actions are intermixed with online I/O and visualization processing.

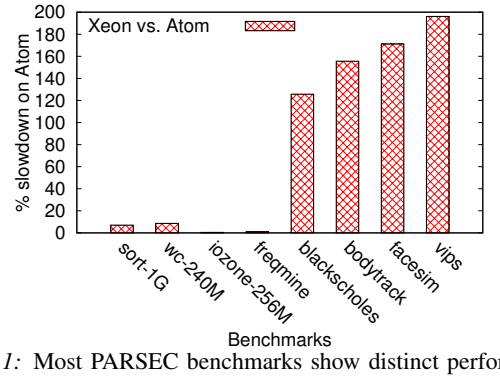

Figure 1: Most PARSEC benchmarks show distinct performance benefit on Xeon cores. Iozone, Hadoop sort and wordcount exhibit similar performance on Xeon and Atom

5.4 Kinship addresses both performance and functional asymmetry

Figure 1 also shows that it is inefficient to fair-share fast cores among all VCPUs since some workloads see little benefit from fast cores. On this basis, we now evaluate the ability of kinship computation to capture different asymmetries and workload combinations. Table 1 lists the tests performed with increasing performance asymmetries and varying degrees of CPU sharing among workloads.

Speed1 demonstrates a scenario with media-processing (VM1) and data-mining (VM2), using corresponding benchmarks from SPEC and PARSEC. These benchmarks vary in their degrees of CPU sensitivity and hence, show less vs. more performance variation when run on asymmetric cores. Montage takes about 30% less time (figure 2(a)) for VM1 compared to stock Xen, without degrading VM2's performance. It also shows highly stable behavior because of its recognition of asymmetry and the predictability of the model that governs scheduling decisions. Omitting graphs for brevity, per-VM improvement can be attributed to Montage doing well for most benchmarks. Vips and dedup, however, show phase changes to slight CPU intensity from time to time, requiring more sophisticated monitoring which is part of our future work. Since the hints given indicate only IO sensitivity (which is true majority of the time), Montage always maps them to slow cores, thus causing minor performance degradation which restricts improvements seen by VM2 for example. Regular Xen, on the other hand, schedules workloads without any regard to asymmetry and exhibits high performance degradation for benchmarks like povray and h264ref while slightly benefiting Vips and dedup at a much greater cost to the overall system. The deviation can be attributed to the fact that every possible schedule for the given VCPUs on the available set of PCPUs is equally likely. This also means, however, that Xen can witness high variability in performance.

Figures 2(b&c) show scheduling performance in the presence of CPU speed asymmetry when the system is oversubscribed. We run two media VMs with standard benchmarks, as shown in Table 1. As seen from the graphs, Montage on average performs better than Xen for most benchmarks and significantly so for some, like h264ref, povray and freqmine. More importantly, even modest performance gains for most benchmarks adds up to a much better overall system efficiency achieved by Montage.

A distinguishing element of kinship is its handling of asymmetries for system resources other than CPU, like cache asymmetry, which has not been the case for previous work. Adding LLC size asymmetry to the *SimulatedWM* platform, Table 1 shows the configuration Speedcache1 used to evaluate CPU speed and socket cache size asymmetry. Small *CacheBuster* fits within a 3MB LLC, while Large *CacheBuster* fits within a 12MB cache. Montage obtains better performance and lower standard deviation (figure 2(d)) than Regular Xen, which, while achieving better than floor performance thanks to its work conserving nature, is hurt by the lack of asymmetry awareness.

Use-case2 from Section 3 (SpeedAes in table 1) demonstrates functional asymmetry, exploiting the AESNI instructions on Xeon processors. AESbench allows us to choose the number of blocks and loops. We keep the number of blocks constant, but vary the loops for big vs. small AES instances. The benchmark checks if a CPU supports AESNI using cpuid. If it does, then the hardware version of AES is executed, else the software version is chosen. The software version is much slower than the hardware counterpart, as expected. We use a 128-bit AES-CTR encryption here. Figures 2(e&f) compare Montage and Regular Xen demonstrating the performance advantages derived from Montage's awareness of functional asymmetry. While the work-conserving nature of Xen is quite useful in improving performance, high deviation is witnessed by Xen due to frequent execution of the aesenc-large (encryption within AESbench) instance on a non-AES supporting CPU, leading to a 2X performance improvement seen by Montage for VM1. The predictability of scheduling in Montage in the absence of oversubscription is an important property establishing its prominence compared to the current Xen scheduler.

5.5 Kinship can adapt to platform evolution

In Section 3, Usecase1 illustrates the intuition behind and definition of kinship with VM1 running *ferret* and *Iozone-512M* on 2VCUs and VM2 running *streamcluster*, *freqmine*, *sort-1G* and *Iozone-512M* on 4VCPUs, respectively. Of these, ferret and streamcluster are compute intensive, freqmine and sort-1G are memory intensive while Iozone is disk intensive. We now evaluate this scenario on different platforms to demonstrate (1) performance improvement achieved on all platforms compared to an asymmetry-unaware Xen hypervisor, (2) the general applicability of kinship-based scheduling across a wide range of asymmetries and their combinations, and (3) portability of the kinship implementation across multiple shared-ISA machines.

Figures 2(g&h) comparing the Montage and Xen schedulers show that appropriate workload scheduling leads to a significant performance improvement for Montage vs. Xen on all platforms. On *QuickIA*, where the VMs have 2 Xeon and 2 Atoms available to share, Montage shows low variability and an overall VM performance improvement of 6% and 12%, respectively, compared to Xen. Montage is comparable to the ceiling numbers shown in the figures. Compute intensive benchmarks like ferret and streamcluster see a stark performance improvement (17% & 200%) from better mappings performed with kinship metric. Since *Dual Xeon* shows less drastic asymmetries, the performance difference is less striking. However, the results still show low variation and better overall performance. The Dom0 in VCPUs in this case share all 6 PCPUs with the two VMs. The slight deviation for MontageXen in Dom2 is a result of inherent IO deviation seen by Iozone. Montage sees similar performance benefits on *SimulatedWM* as well.

Test	Benchmarks (VCPU-wise)	PCPU Config 100%,75%,50%
Speed1	VM1-Media: *povray,h264ref,sc,vips*; VM2-DataMining: *freqmine,dedup*	$\langle 2,2,2 \rangle$
SpeedShare1	VM1-Media: *povray,freqmine,h264ref,vips, sc, dedup*; VM2-Media: *x264,bodytrack,sc,dedup*	$\langle 2,2,2 \rangle$
Speedcache1	VM1: *spin, cachebuster(small)*; VM2: *spin, cachebuster(large)*	$\langle 2,0,2 \rangle$
SpeedAes	VM1: *AESbench (small), swaptions*; VM2: *AESbench (large), dedup*	$\langle 2,0,2 \rangle$

Table 1: PCPU speed, cache and functional asymmetry configurations for *SimulatedWM*. Excess cores are used by Dom0

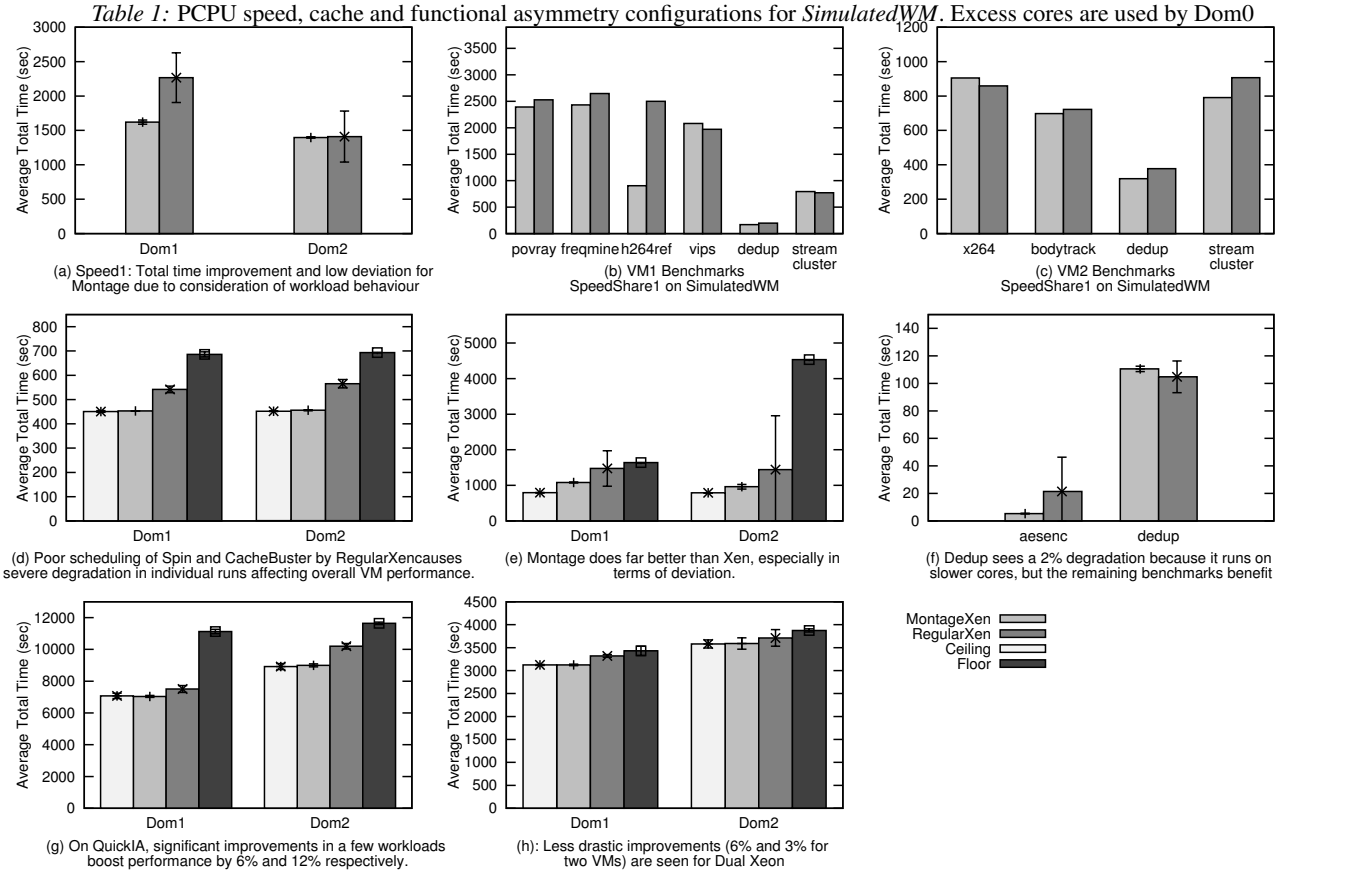

Figure 2: Workload evaluation

5.6 Additional Analysis of Kinship

Scalability to future many-cores: The overheads introduced by the additional computations for kinship in different portions of the hypervisor code are relatively small. The overhead of VM creation and destruction remains within 14-24μs, irrespective of the existing number of VMs in the system. VCPU creation and destruction costs 3-8μs. The cost of assigning a VCPU to a PCPU varies from 1.5-8μs for PCPU numbers between 4 to 12.

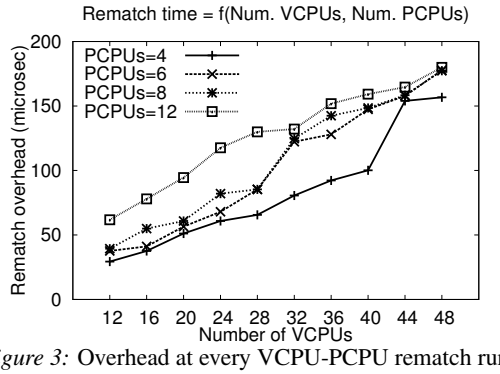

Figure 3: Overhead at every VCPU-PCPU rematch run

However, the most important component of kinship scheduling is the rematching algorithm, which runs periodically to revise VCPU-PCPU mappings based on a change in number of VCPUs or updates to the kinship parameters occurring due to observations or change in platform characteristics. Figure 3 shows the behavior with increasing number of VCPUs at different number of PCPUs in the *SimulatedWM* system. As seen in the figure, this function results in increased overhead with larger numbers of VCPUs or PCPUs. However, reassignment is carried out at most once every four Xen accounting cycles (so $\sim$120msecs) and hence, even with 180$\mu secs$ overhead at 48 VCPUs and 12 PCPUs, it is less than 1% of the accounting cycle time. The graph also indicates that the number of VCPUs becomes the dominating factor the count goes up.

Fault-and-migrate overhead: When a VCPU is run on a core that does not support the full X86 ISA and an unsupported instruction is executed, the hypervisor has to reschedule the VCPU onto a core that supports the instruction. Such micro-scheduling actions occur outside the regular scheduling interval used by the hypervisor or OS and thus, must be highly efficient in order to cope with VMs that frequently use diverse sets of unsupported instructions. On *SimulatedWM*, we observe the overhead for such faulting on unsupported instructions to be 20μs (measured from user level), with an average VCPU migration cost in Xen measured as 495ns. The fault handler simply updates the fault count for the VCPU-PCPU pair and does not cause any kinship modification within the handler. Those are handled at the next rematch. We also note that micro-scheduling [13] has become increasingly cheaper as hard-

ware has evolved, and we expect this trend to continue, in part due to its alternative uses for changing the power states of mobile platforms [4].

Parametric controls for kinship and learning from observations: Equation (2) shows the weights that can be used to modify the effect that each kinship component can have on the overall kinship value. Section 4 talks about how we can observe faults and migrate a VCPU to handle functional asymmetry. As seen from Equation (9), the functional component of kinship between some VCPU and PCPU pair becomes increasingly negative as the fault count increases. At some point, the functional component becomes negative enough to offset any positive value from the performance kinship component due to other resource matches. That's when the VCPU is scheduled on other PCPUs. Since it's a moving time window, the VCPU may eventually get migrated back to the same PCPU if other factors permit.

We execute two VMs with two VCPUs each on the *Dual Xeon* platform to evaluate the effect of weight and functional kinship. VM1 runs $ferret$ and *sse_mat* on its VCPUs while VM2 runs $streamcluster$ and *sse_mat*. *sse_mat* is a microbenchmark that repeatedly runs SSE4.1 based matrix multiplication on really small matrices. Both $ferret$ and $streamcluster$ are more CPU and cache intensive than the *sse_mat* instances. We allow the four VCPUs to run on any of the two 5160 cores that do not support SSE4.1 as well as two out of the four E5440 cores that do support SSE4.1. We do not indicate any category expectations for the VCPUs. Hence, Montage initially picks the two benchmarks from PARSEC, $ferret$ and $streamcluster$ to run on the E5440 cores due to their higher speed factor and larger LLCs. This leaves the 5160 cores, without the SSE4.1 support, for the two instances of *sse_mat* microbenchmark. This will obviously lead to faults since the benchmark expects SSE4.1. Montage maintains this fault count before migrating the VCPUs. However, the VCPU can be migrated back until its kinship value precludes it from doing so, as discussed above.

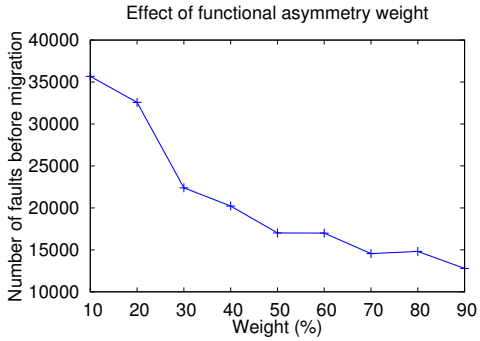

Figure 4: MontageXen migrates a faulting VCPU from the corresponding PCPU and its reactiveness can be modified through the weight assigned to functional portion of kinship

Figure 4 shows the effect of modifying the kinship weights on the maximum fault count seen by VCPU running our SSE microbenchmark. The fault count in this example reaches a high value before the VCPU is permanently migrated due to the performance kinship match which requires a larger negative value as offset. This behavior can be changed using a fault threshold that leads to the VCPU getting classified as a VECTOR VCPU earlier so as to affect the *match factor* in Equation (9). The migration is also delayed by other VCPUs, with higher kinship values that choose their PCPUs before the VCPUs running the SSE benchmarks.

5.7 Discussion

Performance: experiments demonstrate that the kinship formulation successfully parameterizes diverse asymmetry characteristics, thereby creating an evolvable scheduling framework for current and future asymmetric hardware. The low variance in application runtimes and the stable nature of scheduling make its current implementation useful for (barrier-based) HPC codes, but it requires additional sophistication in runtime observation for deployment in general cloud and data center installations.

Implementation challenges: arise from different generation of cores, possibly supporting different revisions of instructions. For example, Xen had to be modified to boot on the *Dual Xeon* platform, since the two sockets supported different revisions of VMCS on the cores. As a result, to run HVM guests on this platform, the VMCS would have to be written and read each time a VCPU is migrated from one generation core to the other, in software. We did not evaluate the additional overheads this would have caused, because vendors would provide hardware or micro-architecture conversions to support asymmetric migration if such platforms were manufactured as mainstream products.

Guest-level scheduling: and asymmetry-aware guests are out of scope for this paper. The issue is avoided in experiments by pinning application threads to VCPUs, but it is important because (1) a smart guest using the Montage collaboration/hint interface could provide hints to the hypervisor and help it switch VCPU-PCPU mappings based on guest-level workload knowledge; and (2) an asymmetry-aware guest would schedule threads on the right kind of cores. Without such functionality, Montage has to rely on observation to detect guest-level phase changes. In such cases, more sophisticated observation capabilities can further help kinship scheduling adapt to dynamic changes in VCPU behavior.

Using hints vs. observation: raises certain trade-offs. The current kinship implementation assumes the correctness of hints. Observation can be used to validate them and/or to report discrepancies to a module outside the performance critical code used by kinship scheduling. We have demonstrated the viability of both by implementing a straightforward monitoring and feedback loop, but as stated earlier, further work is needed to create robust observation-based methods for general server applications.

Limitation: of fault-and-migrate based micro-scheduling is that it requires memory-coherence, so that VCPUs running elsewhere can continue to access the state they previously used. The issue can be addressed by changing programming models to those using CUDA [6] or OpenCL, or by imposing simple restrictions on the 'scope' used for micro-scheduling. For future larger-scale multicore chips, we expect such scopes to align with the multiple coherence domains potentially present on a single chip.

6. RELATED WORK

[10] argues for processor performance asymmetry with fewer fast cores and more smaller, slow cores in the same die area to accelerate the serial phases of parallel applications, and for such platforms, scheduling methods are explored in [12], and [15]. In comparison, kinship addresses both performance and functional asymmetries in platform resources. In addition, kinship considers workload diversity when assigning schedulable entities to underlying platform resources, and it goes beyond specific forms of asymmetry, like those for single parallel codes, to address arbitrary applications seeking their fair shares of diverse cores.

CAMP [19] and HASS [20] map CPU intensive tasks to complex cores using either application signatures or techniques that measure LLC misses and thread level parallelism (parallelism-aware), mapping highly parallel phases to the small cores, but move bottleneck serial phases to the faster core. [9] describes a "bias" metric, based on dynamic observation, to map applications to cores on which the application can execute most efficiently. Kinship extends this con-

cept, allowing each schedulable resource type within the system to have distinct "bias" criteria. [18] suggests 'core fusion' as a principle for using fast vs. slow cores, and the approach in [4] 'mates' a power efficient 'small' core with a 'big' core to maintain power caps and reduce power consumption for client systems, where only a single 'mate' is active at a time. Neither of those two papers explores the scheduling methods needed in such settings.

[7] proposes an asymmetry-aware hypervisor that addresses CPU speed asymmetry and implements fair sharing of fast cores among VMs, occupying fast cores first when performance is the top concern and prioritizing access to rare fast cores based on VM priorities. However, the work does not consider asymmetry in workloads – a prime motivation driving the development of heterogeneous platforms, nor does it address other forms of platform asymmetries like different cache sizes or functionally different units.

[11] extends the Xen scheduler to provide performance asymmetry aware scheduling, monitoring VCPU behavior to correctly match VCPUs to the underlying speed-asymmetric platform. In comparison, kinship-based scheduling addresses a broader range of platform asymmetries in addition to monitoring VCPUs for insights. For accelerator based systems, the coordinated scheduling methods [6] offered for the case of non-overlapping ISAs are complementary to the kinship approach demonstrated in this paper.

Differences in contentiousness vs. sensitivity of applications are formulated in [21] and validated with experimentation that provides methods to measure indicators to classify applications in order to mitigate memory resource contention. Similarly useful for the addition of observation to Montage are the methods in [8]. The symbiotic execution of multiple layers in the software stack in [5, 16] would further enhance convergence to some preferred state, given that different layers will be able to work with more information than otherwise available.

7. CONCLUSIONS AND FUTURE WORK

The *kinship model* and its realization in Montage extend multicore platforms with the first complete representation to address both performance and functional asymmetry, in a manner suitable for a wide variety of asymmetric platforms. Its implementation in the Xen hypervisor enhances credit based scheduling with the kinship equations described in the paper. Experimental evaluations on actual hardware, with a spectrum of asymmetries and with workloads able to exploit them, show that kinship-based scheduling is superior in its ability to map workloads to the types of cores best suited to running them.

Our ongoing research involves implementing methods to enhance the dynamic characterization of workload behavior, in order to provide sophisticated, runtime input for certain parameters included in the kinship equations. The goal is to accurately detect phase changes in workloads, thereby enabling consequent re-mappings of VCPUs to PCPUs. Also of interest are evaluations with asymmetry-aware operating systems and with more sophisticated hint channels, perhaps even allowing operating systems to explicitly state their current requirements.

8. REFERENCES

[1] D. Beaver, S. Kumar, H. C. Li, et al. Finding a needle in haystack: facebook's photo storage. In *OSDI*, Vancouver, BC, Canada, October 2010.

[2] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54, May 2011.

[3] N. Chitlur, G. Srinivasa, S. Hahn, et al. QuickIA: Exploring Heterogeneous Architectures on Real Prototypes. In *HPCA-18*, New Orleans, USA, February 2012.

[4] V. Gupta, P. Brett, D. Koufaty, et al. Heteromates: Providing high dynamic power range on client devices using heterogeneous core groups. In *IGCC*, San Jose, USA, 2012.

[5] V. Gupta, R. Knauerhase, and K. Schwan. Attaining system performance points: Revisiting the end-to-end argument in system design for heterogeneous many-core systems. *SIGOPS OSR*, 2011.

[6] V. Gupta, K. Schwan, N. Tolia, et al. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In *USENIX ATC*, Portland, USA, June 2011.

[7] V. Kazempour, A. Kamali, and A. Fedorova. AASH: an asymmetry-aware scheduler for hypervisors. In *VEE*, Pittsburgh, USA, 2010.

[8] R. Knauerhase, P. Brett, B. Hohlt, et al. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3), 2008.

[9] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *EuroSys*, Paris, France, Apr 2010.

[10] R. Kumar, D. M. Tullsenand, P. Ranganathan, et al. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA*, München, Germany, 2004.

[11] Y. Kwon, C. Kim, S. Maeng, et al. Virtualizing performance asymmetric multi-core systems. In *ISCA*, San Jose, USA, June 2011.

[12] N. Lakshminarayana, S. Rao, and H. Kim. Asymmetry aware scheduling algorithms for asymmetric multiprocessors. In *WIOSCA*, Beijing, China, June 2008.

[13] M. Lee and K. Schwan. Region scheduling: efficiently using the cache architectures via page-level affinity. In *ASPLOS*, London, England, UK, 2012.

[14] T. Li, D. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *PPoPP*, Raleigh, USA, 2009.

[15] T. Li, P. Brett, R. Knauerhase, et al. Operating system support for overlapping-isa heterogeneous multi-core architectures. In *HPCA*, Bangalore, India, January 2010.

[16] Z. Li, Y. Bai, H. Zhang, et al. Affinity-aware dynamic pinning scheduling for virtual machines. In *CloudCom*, Indianapolis, USA, December 2010.

[17] E. Marcial. The ice financial application. `https://www.theice.com/homepage.jhtml`, August 2010. Private Communication.

[18] S. Panneerselvam and M. M. Swift. Chameleon: operating system support for dynamic processors. In *ASPLOS*, London, UK, 2012.

[19] J. C. Saez, M. Prieto, A. Fedorova, et al. A comprehensive scheduler for asymmetric multicore systems. In *EuroSys*, Paris, France, April 2010.

[20] J. C. Saez, D. Shelepov, A. Fedorova, et al. Leveraging workload diversity through os scheduling to maximize performance on single-isa heterogeneous multicore systems. *JPDC*, 71, January 2011.

[21] L. Tang, J. Mars, and M. L. Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *EXADAPT*, San Jose, USA, June 2011.

[22] F. Zheng, H. Abbasi, C. Docan, et al. Predata - preparatory data analytics on peta-scale machines. In *IPDPS*, Atlanta, USA, 2010.