Scheduling on Heterogeneous Multicore Processors Using Architectural Signatures

Daniel Shelepov
School of Computing Science
Simon Fraser University
Vancouver, Canada
dsa5@cs.sfu.ca

Alexandra Fedorova
School of Computing Science
Simon Fraser University
Vancouver, Canada
fedorova@cs.sfu.ca

ABSTRACT
Heterogeneous multicore architectures promise greater energy/area efficiency than their homogeneous counterparts. This efficiency can only be realized, however, if the operating system assigns applications to appropriate cores based on their architectural properties. While several such heterogeneity-aware algorithms were proposed in the past, they were not meant to scale to a large number of cores and assumed long-lived threads due to reliance on continuous performance monitoring of threads for core assignment purposes. We propose a scheme that does not rely on dynamic performance monitoring. Instead, the information needed to make an appropriate core assignment decision is provided with the job itself. This information is presented as an architectural signature of the application, and is composed of certain microarchitecture-independent characteristics. An architectural signature is generated offline and can be embedded in the application binary. In this paper we describe our preliminary work on architectural signature framework. We present architectural signatures that predict application's sensitivity to variations in core clock speed. We evaluate a scheduler prototype that uses these signatures for scheduling on a heterogeneous system composed of cores with various clock frequencies.

Categories and Subject Descriptors
D.4 [**Operating Systems**]: Organization and Design, Process Management

General Terms
Algorithms, Measurement, Performance, Experimentation

Keywords
Heterogeneous, Scheduling, DVFS, Architectural signatures, Microarchitecture-independent, Sensitivity to changes

---

This research is supported by an NSERC USRA grant and by Sun Microsystems.

1. INTRODUCTION
Future multicore and many-core processors will consist of heterogeneous cores that may expose a common ISA but differ in features, size, performance and energy consumption [3,6,14]. A single processor will contain many small simple cores, and several larger complex cores. Simple cores will be scalar, in-order and might have a smaller cache and lower clock frequency. Complex cores will be super-scalar, and may be equipped with high-performance features such as out-of-order instruction scheduling, aggressive pre-fetching or a vector unit. Complex cores will be larger and consume significantly more power. Heterogeneous architectures are motivated by their potential to achieve a higher performance per watt than comparable homogeneous systems [9,10], because each application can run on a core that best suits its architectural properties.

To realize this potential, however, the operating system must be heterogeneity aware. That is, it must assign applications to run on appropriate cores. Consider a workload consisting of a scientific application characterized by high instruction level parallelism (ILP), and a memory-bound job such as transaction processing in a database. The scientific application will be executing significantly faster on a complex core, whereas the database application might show comparable performance on both types of cores. In this scenario, assigning threads to cores that are appropriate for them (the scientific application to a complex core and the database application to a simple core) will achieve significant power savings. Being able to make such intelligent decisions at runtime is the goal of heterogeneity-aware scheduling.

While heterogeneity-aware scheduling algorithms were proposed in the past [4,7,9,10], they were targeted at small-scale multicore systems and assumed long-lived threads. As we explain in the next section, they relied on continuous performance monitoring of threads to determine the optimal assignment of threads to cores. As the number of cores per chip increases [6], dynamic monitoring may become too time-consuming and impractical.

We propose a new approach to scheduling on heterogeneous multicore systems based on *architectural signatures*. An architectural signature is a short summary of the application's architectural properties, such as its memory-boundedness, available ILP, etc. The signature is generated off-line and embedded in the application binary. The idea is that the

operating system will determine the best thread-to-core assignment simply by examining this signature, and *without* any dynamic monitoring of applications' performance.

In this study we begin exploring the signature approach to scheduling. We describe how we generated architectural signatures that predict application sensitivity to variations in clock speed frequency. These signatures can be used by the scheduler on a heterogeneous system where cores differ in their clock frequency. The scheduler would assign insensitive applications to low-frequency cores and sensitive applications to high-frequency cores and thus maximize performance per watt. We have built and evaluated a user-mode scheduler prototype that determines appropriate cores for threads using signatures. We found that architectural signatures are good predictors of sensitivity to clock frequency and that the scheduler can improve performance using them. In the future we plan to extend our architectural signature framework to cover other architectural parameters.

A potential drawback of the signature-based scheduling is that it does not take into account dynamic phase behavior of the application [12]. In our initial design, there is only one signature associated with the application, while in reality the application may contain many phases that have different architectural properties and require different thread-to-core mappings. Another limitation is that signatures are difficult to adapt to varying input sets, which can sometimes drastically change program performance. Potential benefits of this approach, on the other hand, are (1) simpler implementation of the scheduler, (2) greater scalability potential on many-core systems, and (3) support for short-lived threads. It is the goal of our research to investigate whether the simplicity and scalability of the signature-based approach outweigh its drawbacks.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 presents the methodology for generation of architectural signatures and describes how they are used in the scheduler. Section 4 presents preliminary evaluation of our algorithm via a user-level prototype on a heterogeneous multicore system. Section 5 summarizes our work and outlines the plans for future research.

2. RELATED WORK

The two most prominent heterogeneity-aware scheduling algorithms were described by Becchi [4] and Kumar [10]. Becchi's algorithm relied on continuous monitoring of performance of each thread on each type of core to determine the best thread-to-core assignment for the system. Kumar's algorithm used a similar approach as well as another technique, where selected thread-to-core assignments were tried by the operating system and then the best-performing assignment was used thereafter.

While these algorithms demonstrated improved performance over a heterogeneity-agnostic scheduler, they have several drawbacks. When the number of different core types is large (as may be the case on future many-core systems), monitoring of individual threads or thread subsets may become infeasible. The operating system will have to track too much information, which may hurt performance. Furthermore, the amount of time used for monitoring (before the optimal thread-to-core assignment is determined) may become long, and so until the monitoring is complete the system runs sub-optimally. Besides, if threads are short-lived and determining the optimal core assignment takes a long time, threads they may terminate before the scheduler has had a chance to learn a good assignment.

There are several trade-offs to consider when comparing the dynamic approaches used by Becchi and Kumar and our signature-based approach. Dynamic approaches can adapt to program phase changes (because they monitor performance continuously) and may determine the optimal thread-to-core assignment more efficiently, since they rely on real performance measurements as opposed to modeling, which is inevitable with a signature-based approach. On the other hand, dynamic approaches are vulnerable to the drawbacks that we discussed above. Our goal is to evaluate these tradeoffs and to develop a better understanding of scheduling on heterogeneous multicore systems.

3. METHODOLOGY

3.1 General Overview

A heterogeneous processor contains cores with different capabilities. At the same time, the workload running on these cores also might and often will be heterogeneous, meaning that different applications will have different resource demands. A scheduler managing tasks on such a heterogeneous system will improve system performance if it is able to align the workload with the cores in a way that will maximize the utilization of resources on each core.

In essence, what the scheduler needs to know at runtime is how each job's performance changes depending on the availability of architectural resources. The resources considered might be cache size, clock frequency, issue width, a FPU, a vector unit, or an intelligent branch predictor. One way to acquire this information offline is to run the job on cores that differ along those criteria and store the resulting performance data in a table- or matrix-like form. This approach is unwieldy, because it requires either many different cores to be available for testing, or a simulator. In both cases, as the number of possible permutations of different characteristics becomes large, this method becomes infeasible due to cost or time constraints. Another way is to let the programmer specify the architectural signature of the job. This is unrealistic for most applications, because it requires very deep understanding of program behaviour, often at a level much lower than programmers are comfortable with. A third strategy is to run a program, and based on *what* it does figure out what resources it might benefit from. This is the path we take.

A well-known way of tracing the behaviour of a program is through binary instrumentation, which allows arbitrary code to be executed during pre-defined points in program as it runs. In our research we used Pin, a binary instrumentation framework from Intel [11]. Pin is non-invasive, in that it does not require any modifications of the target program for instrumentation purposes. Besides providing many common binary instrumentation tools, Pin allows custom toolkits to be built on top of it, and one of such custom toolkits is Microarchitecture-Independent Workload Characterization (MICA) by Hoste and Eechhout [8]. MICA uses Pin to collect metrics such as opcode (instruction) mix, memory access patterns, register access patterns, the amount of inherent ILP and others. Although these

Table 1. Machines used for gathering elasticity data

#	CPUs	Caches	DVFS settings used (GHz)
1	Intel Xeon E5320 (Clovertown), 4 cores	L1I 4x32K (8-way), L1D 4x32K (8-way), L2 2x4MB (16-way)	1.6, 1.87
2	Intel Pentium 4 (Northwood), 2 cores	L1D 2x8K (4-way), L2 2x512K (8-way)	2.1, 2.8
3	Intel Xeon X5365 (Clovertown), 8 cores	L1I 8x32K (8-way), L1D 8x32K (8-way), L2 4x4MB (16-way)	2, 2.33, 2.67, 3

**All machines run Linux 2.6.xx (Redhat)*

are specific to binaries for x86 architecture, they are *micro*architecture-independent, meaning that they do not imply or rely on specific core characteristics. Thus, we need only one system to generate all the data required for producing an architectural signature for an application, and that signature will be useful on any x86 system where that binary runs.

The binary instrumentation process itself is notoriously slow, and we have seen run time increases on the order of tens of thousand percent. Making a complete run of a program is likely unacceptable as a means of generating an architectural stamp, but it is possible that instrumenting over intelligently picked intervals during the run can significantly cut down the time required with little loss of accuracy. This problem, however, is outside the scope of the paper.

The only remaining question is how to leverage microarchitecture-independent job characteristics provided by MICA to predict whether the job benefits from a larger cache, an out-of-scalar core, etc. Each of such core features or parameters requires its own separate prediction model. In this paper, we study the architectural signature framework in the context of varying clock speed (i.e. frequency). Our goal is to generate architectural signatures that will help the scheduler predict if an application's performance is sensitive to the clock frequency of the processor. We chose to study clock frequency, because we can set up a real heterogeneous system using dynamic voltage and frequency scaling (DVFS) feature on general-purpose multicore machines. Studying other architectural parameters is the subject of future work.

3.2 Measuring Clock Speed Sensitivity

In general, program performance does not scale linearly with clock frequency of the underlying core. This is because besides performing computational tasks, a process also makes memory requests, which take a relatively long time and are somewhat independent of the core's speed. Since different programs exhibit different memory access behaviour, their performance (which in this paper we tie to completion time) scales with clock frequency at different rates. This observation is key to differentiating tasks according to their clock speed sensitivity.

Our first step is to define a metric for clock speed sensitivity. Towards that end, we picked a set of 25 benchmarks from SPEC CPU2000 suite[1] and ran those individually using different

[1] All SPEC CPU2000 benchmarks were utilized, except *galgel*, which we could not get to run correctly on our test platform.

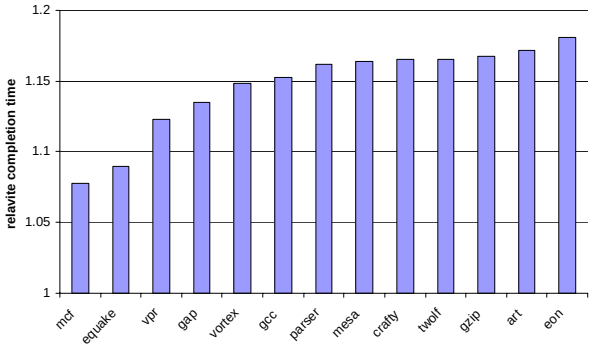

Figure 1. Relative slowdown of selected benchmarks when run on 1.6GHz compared to 1.87GHz

frequency settings on our DVFS-enabled machines (see Table 1 for specifications of these machines). DVFS presents an especially convenient way to measure effects of clock speed changes, because any change in completion time can be attributed exclusively to a difference in core frequency.

As expected, different benchmarks exhibited different speedups when running on a faster core. Figure 1 illustrates the increase in completion time of some benchmarks when run on 1.6GHz as opposed to 1.87GHz on machine 1. Benchmarks such as *eon* and *art* react strongly to changes in frequency, while *mcf* and *equake* do not. To put a number on this behaviour, we have borrowed the arc elasticity formula from economics to define *clock speed elasticity of completion time*, which will be used as our clock speed sensitivity metric. Elasticity of a function is its sensitivity to a change in a variable. Rigorously speaking, if T is completion time and F is clock frequency, we have:

$$E_{T,F} = \frac{T_2 - T_1}{F_2 - F_1} * \frac{F_1 + F_2}{T_1 + T_2}$$

which is calculated on some region $[F_1, F_2]$. In plain terms, we can view elasticity as the ratio of a relative change in T over a relative change in F.

As an example, let us say that *vpr* completes in 58.17 seconds running on a core at 1.87GHz, and takes 65.33 seconds when the frequency is set at 1.6GHz. The elasticity on [1.6, 1.87] is then calculated as follows:

$$E = \frac{65.33 - 58.17}{1.6 - 1.87} * \frac{1.87 + 1.6}{65.33 + 58.17} \approx -0.75$$

Also note that elasticity generally ranges from $-\infty$ to $+\infty$, with negative values representing an inverse relationship that we expect to see in our scenario (completion time decreases following increases in clock speed). Higher magnitude indicates higher sensitivity of performance to changes in clock speed, and vice versa (elasticity of 0 would indicate that performance does not at all depend on clock speed over the range specified). As a base point, an elasticity of -1 means that a change of clock speed by a certain factor will decrease the completion time by the same factor.

Using this definition, we have gathered the elasticity data, a sampling of which is available in Table 2.

Of interest are the results for machine 2, which has a significantly smaller L2 cache than the other two machines. We

Table 2. Measured elasticity values of selected benchmarks

	machine 1, [1.6, 1.87GHz]	machine 2, [2.1, 2.8GHz]	machine 3, (average)
mcf	-0.486	-0.821	-0.512
equake	-0.557	-0.808	-0.499
vpr	-0.752	-0.776	-0.727
gap	-0.818	-0.790	-0.858
vortex	-0.896	-0.789	-0.895
gcc	-0.917	-0.798	-0.937
parser	-0.971	-0.797	-0.968
mesa	-0.981	-0.799	-0.968
crafty	-0.988	-0.814	-1.004
twolf	-0.991	-0.780	-1.005
gzip	-1.000	-0.774	-0.989
art	-1.023	-0.785	-1.007
eon	-1.074	-0.904	-1.000

**Values for machine 3 were calculated by averaging elasticity values on regions [2, 2.33GHz], [2, 2.33GHz] and [2, 2.33GHz].*

can see that a smaller cache causes elasticity values to converge. This indicates that clock speed sensitivity is heavily dependent on cache size, a microarchitecure-dependent variable. Therefore, in our approach we will calculate several sensitivity values for common cache configurations and have the OS select the appropriate one at scheduling time. If we have a heterogeneous system where the cores differ only in their frequencies, the OS will pick the sensitivity value for the cache size used on these cores. If, however, we use a heterogeneous system where the cores also differ in their cache sizes, the OS will first try to pick a core with an appropriate cache size for a job to run on, and then optimize the clock speed. Otherwise any clock-speed benefit might be nullified by a severely limiting cache size.

More details on this scheme will be provided in the following sections, but for now the goal is to construct a model for predicting clock speed sensitivity for different cache sizes, using microarchitecture-independent job characteristics gathered with MICA.

3.3 Predicting Clock Speed Sensitivity

As we have mentioned in Section 3.2, memory access patterns of a job can be used to explain its clock speed sensitivity. One of the most easily interpreted memory access metrics is the cache miss rate. Intuitively, the more misses there are in a cache, the more time is the CPU stalled, and the less benefit there is in having a higher core frequency. L2 miss rate can be especially telling, due to ever-growing performance gap between the CPU and main memory. This is why we chose L2 cache miss rate estimation as our means of calculating clock speed elasticity.

Figure 2 shows how we derive elasticity values that are presented to the OS scheduler. As shown, elasticity is computed as the linear function of the L2 cache miss rates. The microarchitecture-independent linear function is derived offline using linear regression and L2 cache miss rates are also estimated offline using the microarchitecture-independent metrics collected with MICA. We now explain how we estimate the L2 cache miss rates.

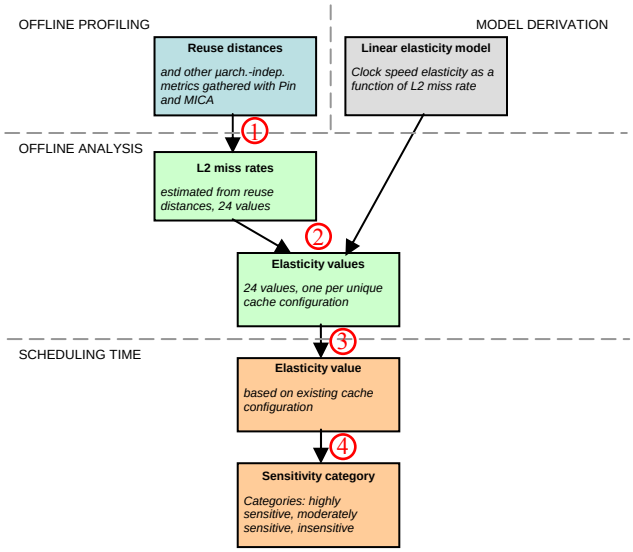

Figure 2. Dataflow diagram of our scheduling framework: (1) L2 miss rates for common cache configurations are estimated using reuse distances (2) For each cache configuration, an elasticity value is estimated from L2 miss rate (3) Elasticities are made available to the scheduler (4) The scheduler places the program in a sensitivity category based on elasticity

Estimating cache miss rates without directly simulating a memory hierarchy is a challenging problem. Berg and Hagersten suggest tracking *reuse distances* of memory references to estimate miss rates [5]. We opt for a similar approach with some modifications, necessitated by the fact that reuse distances computed by MICA are defined somewhat differently than those used by Berg and Hagersten[2].

For any memory address we consider, let's call the previous access to that location the *initial reference* and the current access to it the *final reference.* The reuse distance of the current access is then the number of distinct memory locations touched between the initial and the final references. A large reuse distance indicates a high chance of a cache miss, because many other cache blocks would have been requested and possibly brought into the cache since the initial reference, and the chance that the cache block we need is still resident in the cache is decreased.

In MICA the reuse distance is calculated according to the number of unique 64-byte blocks touched since the last reference to the same block. MICA is able to categorize memory accesses into 20 buckets according to their reuse distance. There is one bucket for reuse distance of 0 (meaning that the last block touched was the same block), one for cold references (the block was touched for the first time since the start of execution), one for distances larger than 2^17, and seventeen buckets for distances between consecutive powers of

[2] MICA calculates a reuse distance based on the number of *unique* intervening references before the reuse of the same location, while Berg and Hagersten count *all* intervening references.

2 from 0 to 17. Each memory reference is reported in exactly one bucket, so at the end of the run (or an interval), we are able to see how many memory accesses fell into each bucket:

B_∞ = # of cold references

B_0 = # of accesses with reuse distance 0

$B_{(131073, \infty)}$ = # of accesses with reuse distance more than 2^{17}

$B_{(1, 2)}$, $B_{(3, 4)}$, $B_{(5, 8)}$,.. $B_{(65537, 131072)}$ – buckets for references with reuse distances that fall into corresponding ranges

Our approach is then to estimate the number of misses based on bucket values:

$m(B)$ = # of misses generated by accesses from bucket B

References from B_0 are unlikely to trigger any cache misses, because a reuse distance of 0 implies that there was no opportunity to evict the cache line. It can still be evicted by a co-runner thread, in a context switch or by an overzealous prefetcher, but we ignore these cases for the sake of simplicity. Instead, we consider accesses from B_0 to always hit in the cache.

Similarly, we consider references in B_∞ to be compulsory cache misses. Thus we have

$m(B_0) = 0$

$m(B_\infty) = B_\infty$

The other 18 buckets are left to consider. We build on the assumption that memory accesses that have similar reuse distance should have similar chances of triggering a cache miss:

$$m(B) = B * p(B) \tag{1}$$

where $p(B)$ is the bucket-specific cache miss rate.

We don't want to deal with ranges of reuse distances, so for the sake of simplicity, we consider all references in a bucket $B_{(x, y)}$ to have the same reuse distance, which we define as

$$r(B_{(x,y)}) = 4x/3 \tag{2}$$

For example, $r(B_{(65537, 131072)}) = 4 / 3 * 65537 = 87164$.

As we consider the possible values of $p(B)$, the problem of different cache sizes comes into view. A memory read with a certain reuse distance might likely to trigger a miss in a cache of one size, but not to trigger it in a larger cache. After an investigation, we were convinced that the size, as well as degree of associativity of the L2 cache, had to be considered for further analysis. Thus our model splits into several branches, each assuming a common set associativity (we chose powers of two from 4 to 32) and a common L2 cache size (powers of two from 512K to 16MB). Final sensitivities for all these variants (there are 24 of them) are made available to the scheduler, so it can pick the appropriate sensitivity value at scheduling time.

After we know the cache size and set associativity, we can make estimations of $p(B)$ values. Let's consider a memory access with reuse distance r and an S-way associative cache of N lines with LRU replacement. The cache line we're interested in has S-1 neighbours in its set, and at the time of the initial reference, it is the most recently used cache line of the set. Notice that if there are requests to S or more unique locations that map into this set, our cache line must be evicted as it will be the least recently used by the Sth request.

The total number of sets in the cache is N / S. Assuming uniform distribution of memory references across the address space, the probability of a request being mapped into any particular set is $1 / (N / S) = S / N$. The probability that it maps into any other set is similarly $(N – S) / N$. With this knowledge, we can calculate the probability of exactly i requests out of r falling into the same set:

$$\binom{r}{i} * \left(\frac{S}{N}\right)^i * \left(\frac{N-S}{N}\right)^{r-i} \tag{3}$$

By evaluating expressions (3) for all i's from 0 to S-1, we can find the probability of having fewer than S requests out of total r mapped into any one set with this formula:

$$\sum_{i=0}^{S-1}\left[\binom{r}{i} * \left(\frac{S}{N}\right)^i * \left(\frac{N-S}{N}\right)^{r-i}\right] \tag{4}$$

The complement of that, 1 – (4) is the probability of there being S or more requests in the set, or, in other words, the probability of our cache line being evicted after r unique cache blocks are requested. Using this fact, we can find $p(B)$ for a given cache configuration:

$$p(B) = 1 - \sum_{i=0}^{S-1}\left[\binom{r(B)}{i} * \left(\frac{S}{N}\right)^i * \left(\frac{N-S}{N}\right)^{r(B)-i}\right] \tag{5}$$

Now it is easy to find $m(B)$ for all buckets, using equations (1), (2) and (5). Adding together all twenty $m(B)$ values allows us to estimate the total number of misses occurring.

We have also found it beneficial to scale the resulting sum by a certain factor L, which is defined as a function of the cache size (N) and the total working set (W) of the program:

$$L = \begin{cases} W > N : \dfrac{W-N}{W} \\ W \le N : 0 \end{cases} \tag{6}$$

The total working set is simply how much memory the application ever touches during runtime. It this case, W is the number of unique 64-byte blocks touched by the application, and is available as one of MICA metrics.

Scaling by L ensures that the miss rate is going to further decrease as the working set size decreases (and the miss rate will drop to zero whenever the whole working set fits completely into the cache).

By dividing the result by the total number of retired instructions, we arrive at the final L2 miss rate value:

$$\text{miss rate} = \frac{\text{L} * \sum \text{m(B)}}{\#\text{ of retired instructions}}$$

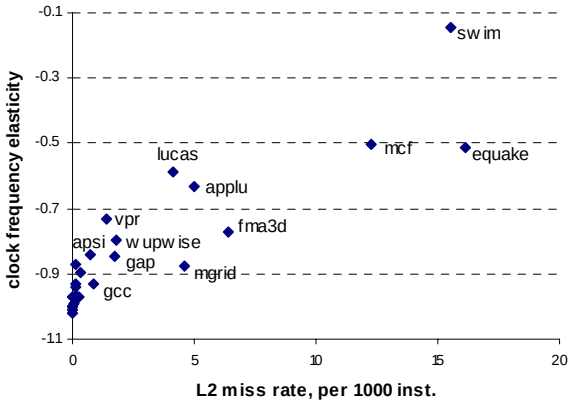

Figure 3. Estimated L2 cache miss rate vs. measured elasticity (assuming a 4MB 16-way associative cache)

Although this estimation assumes an LRU cache, caches with random replacement policies are known to have miss rates on average within a small factor of comparable LRU cache miss rates [13].

The metric used is fairly naïve in that it does not take into account some of the more complicated cache behaviour, such as prefetching. In fact, after collecting miss rate data explicitly on one of our test machines, we have found that in several cases the actual miss rates observed are significantly lower than estimated. Nevertheless, the metric does correlate quite well with observed clock speed elasticity, which is its primary purpose (see Figure 3).

We use linear regression to construct a function describing the relationship between the L2 cache miss rate and elasticity. Constructed once, this function can be used without any further direct elasticity measurements. Instead we use it to estimate elasticity for particular cache parameters. Figure 4 compares estimated elasticities with those that were actually observed for a 16-way 4MB cache. The average absolute error in estimated elasticity values was about 0.075. In general, the metric does a very good job at separating highly elastic applications from inelastic ones.

We are still investigating whether a single miss rate vs. elasticity function could be used across all caches, or if it needs to be constructed for each cache configuration separately.

3.4 Using Architectural Signatures in the Scheduler

As shown in Figure 2, each application has embedded in its binary 24 elasticity values, one for each cache configuration with sizes ranging from 512KB to 16MB and associativities ranging from 4 to 32. To pick the right value, the scheduler has to know the cache size and set associativity before selecting the correct value. This information can be obtained by the scheduler by reading a model-specific register (MSR) on the underlying processing cores.

The scheduler places applications into three categories according to their elasticities: highly sensitive, moderately sensitive and insensitive. Jobs that have elasticity smaller than

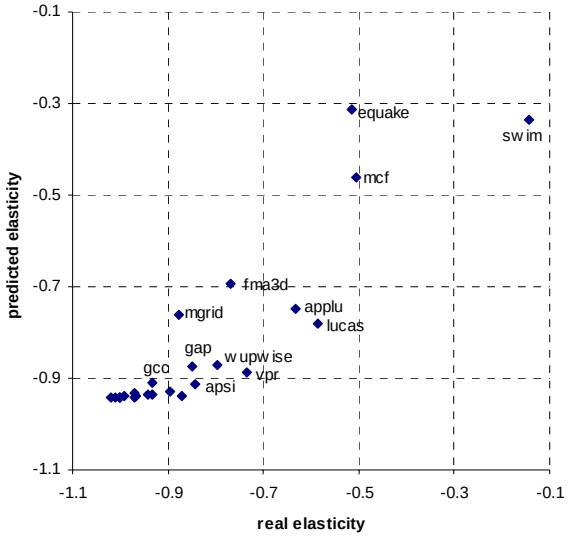

Figure 4. Predicted elasticity vs. measured elasticity, assuming a 4MB 16-way associative cache

-0.9 are considered highly sensitive, jobs with elasticity larger than -0.75 are considered insensitive, and the rest are considered moderately sensitive. This ternary scheme is simple and retains enough information to aid scheduling. Although the boundaries between the categories are determined mostly arbitrarily at this point, this division could be made more meaningful given statistical information on distribution of elasticities across applications. Alternatively, the scheduler could determine appropriate placement at scheduling time. In any event, for the purposes of testing we selected the scheme described, and resulting categories for our benchmarks are summarized in Table 3. While in this paper we consider only clock frequency elasticity, in the future we envision a framework where application signatures are used to estimate the performance effect of several core features. Besides clock frequency, these might be cache size, issue width, the number of present FPUs, and others. Some of these parameters are actually more critical to application performance than clock speed. In fact, the

Table 3. Categories of clock speed sensitivities for benchmarks assuming a 4MB 16-way associative cache (high – highly sensitive, medium – moderately sensitive, low – insensitive). For reference, actual sensitivity categories are noted in parentheses for benchmarks where predicted categories were incorrect.

ammp	high	*lucas*	medium (low)
applu	low	*mcf*	low
apsi	high (medium)	*mesa*	high
art	high	*mgrid*	medium
bzip2	high	*parser*	high
crafty	high	*perlbmk*	high
eon	high	*sixtrack*	high
equake	low	*swim*	low
facerec	high (medium)	*twolf*	high
fma3d	low (medium)	*vortex*	high (medium)
gap	medium	*vpr*	medium (low)
gcc	high	*wupwise*	medium
gzip	high		

evidence is strong that cache size is generally a more important factor than frequency. Therefore, it is reasonable to expect the scheduler to first optimize core assignment by cache size and only then by clock speed. This allows us to assume that the cache size and set-associativity are known before clock speed optimizing scheduling stage is entered.

We hope to present results of this more general scheduling infrastructure in future work; meanwhile, in this paper we show the evaluation results of the signature-based scheme that is restricted to clock frequency elasticity.

4. EVALUATION

In this section we explore the benefits of heterogeneity-aware scheduling based on clock speed sensitivity. The experiments we performed compare the default Linux scheduler with a heterogeneity-aware user-mode scheduler prototype that we have developed.

Our heterogeneous machine (machine 3 from Table 1) is an 8-core Intel Quad system. There are four chips, each housing a pair of cores and a shared L2 cache. We have used only one core per chip to prevent cache interference effects. On these four cores we ran 8 SPEC CPU2000 benchmarks. We provide heterogeneity by having the cores run at different frequencies, as allowed by DVFS settings (we report on specific frequency settings further in this section). Benchmarks are run continuously, that is, after any benchmark terminates, it is immediately restarted. We compare average completion times of benchmarks on a default Linux scheduler (version 2.6.18) vs. our prototype.

The default scheduler utilizes *natural affinity*, meaning that a thread is migrated to a different core generally only if there is a workload imbalance. Other than that, the scheduler is free to assign any thread to any core out of the four that we had enabled for the experiment.

Our prototype works by statically binding benchmarks to cores while maintaining a set of invariants:

a) All four cores have equal loads at all times.

b) Benchmarks marked moderately sensitive always run on cores faster than or just as fast as cores running insensitive benchmarks.

c) Benchmarks marked moderately sensitive always run on cores slower than or just as fast as cores running highly sensitive benchmarks.

d) Every benchmark was bound to as many cores as possible without compromising invariants a-c.

A binding scheme honoring these invariants was determined in advance and enforced using Linux *taskset* utility. Benchmarks were free to migrate between the cores to which they were bound as determined by the default scheduler.

The setup allows benchmarks to run under moderately realistic conditions, exposed to adverse effects such as bus contention, which are not taken into account by the sensitivity prediction framework. Although benchmarks did exhibit clock sensitivity patterns that we expected, resource-sharing definitely caused perceivable noise. For example, in one of our tests (not reported below), we saw *swim* showing higher clock speed sensitivity than would be suggested by its extremely low elasticity (-0.15). Instead, *swim* behaved comparably to *mcf,* which has an elasticity of about -0.50. Deconstructing and predicting these effects might be an interesting area for future investigation, but we didn't explore them any further in this round of experiments.

We evaluated our scheduler using three workloads: a highly heterogeneous workload, a balanced workload and a uniform workload. We expect to see the most performance gain with the heterogeneous workload and the least gain with the uniform workload.

4.1 Highly heterogeneous workload

In this experiment, we wished to explore the limits on performance gains achievable with the heterogeneity-aware scheduler on our heterogeneous (in terms of clock speed) system. To this end, we have chosen two of the most sensitive benchmarks (*eon* and *crafty*) and two that are very insensitive (*mcf* and *equake*). We ran two copies of each simultaneously for an average load of two threads per core. On the system side, we provided two cores running at 2GHz and two cores running at 3GHz. We ran the load for at least 2000 seconds on each scheduler to get the average completion times shown in Figure 5. Under the heterogeneity-aware scheduler, *eon* has shown a 19.4% decrease in completion time, *crafty* has shown a 19.9% decrease, *mcf* has an increase of 15.6% and *equake* – an increase of 12.3%. Aggregate completion time has decreased by 4.3%. It was calculated by taking a geometric average of benchmark completion times normalized to their completion times under the default scheduler.

Had the range of allowed DVFS settings been more significant, the average reduction in completion time would be more dramatic. We estimate that with a pairing 1.5 – 3GHz, the aggregate completion time decrease could almost double, and with pairing 1.0 – 3GHz, the decrease could more than triple the decrease that we observed in our experiment. Thus clock speed sensitivity aware scheduling would be more important if the cores significantly differed in frequency. Such larger frequency ranges may be found in future many-core systems [6].

4.2 Typical workload

In this experiment, we wanted to test the scheduler under a more realistic *balanced* workload. We picked 8 different benchmarks: three insensitive (*mcf, fma3d* and *equake*), two highly sensitive (*gcc* and *eon*), and three moderately sensitive (*gap, wupwise and lucas*). Furthermore, two benchmarks were miscategorized by our sensitivity prediction model (according to the real elasticity we observed, *fma3d* should be moderately sensitive, and *lucas* insensitive), which is also a realistic occurrence. This time we had one core running at each of 2.0, 2.33, 2.67 and 3.0GHz. The results of this experiment are shown in Figure 6. We see that the aggregate completion time has decreased by 2.7% when using heterogeneity-aware scheduling, which shows that even under realistic milder conditions than in the first experiment, there is still a good opportunity for optimization.

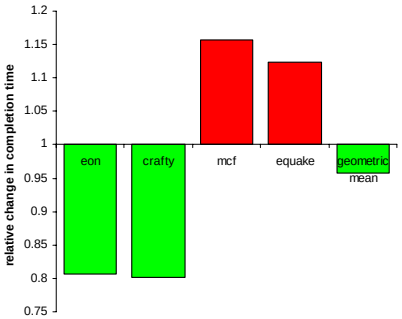

Figure 5. Heterogeneity-aware scheduler performance compared to default scheduler under a very heterogeneous workload

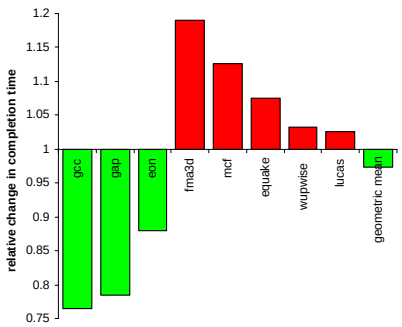

Figure 6. Heterogeneity-aware scheduler performance compared to default scheduler under a balanced workload

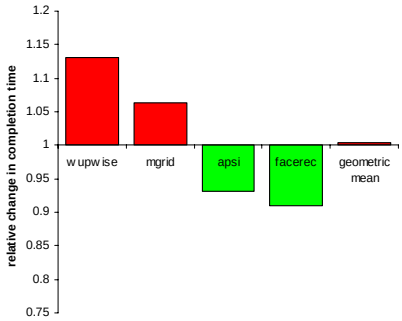

Figure 7. Heterogeneity-aware scheduler performance compared to default scheduler under a homogeneous workload

4.3 Uniform workload

Finally, we wanted to test whether our heterogeneity-aware scheduler causes the performance to degrade when there is little or no opportunity for optimization. For this setup, we have again used the cores clocked at 2.0, 2.33, 2.67 and 3.0GHz. We also used four different benchmarks that have similar elasticity (ranging from -0.796 to -0.877). Two of them (*wupwise*, *mgrid*) are categorized as moderately sensitive, and the other (*apsi*, *facerec*) as highly sensitive. Each benchmark was run in two instances at any given moment. Our scheduler attempted to optimize core assignment by moving moderately sensitive jobs to slower cores, but the performance loss that occurred as a result was comparable to gains observed by highly sensitive benchmarks, resulting in a marginal net slowdown of 0.4% (see Figure 7). This suggests that a heterogeneity-aware scheduler is unlikely to seriously hurt overall performance of a homogenous workload. That said, assignment inversion problems may occur when a more sensitive job is classified as less sensitive than another.

5. SUMMARY

We presented a signature-based framework for scheduling on heterogeneous multicore systems. As a preliminary evaluation of the architectural signature framework we derived signatures for prediction of sensitivity to changes in core frequency, and demonstrated that the signatures are good predictors. We also demonstrated (via a user-level prototype) that a signature-based scheduling algorithm improves performance over a heterogeneity-agnostic scheduler, and we expect the performance benefit to grow as cores become more differentiated from one another, and as we add more core features into the model.

Our plans for future work include (1) implementation and evaluation of a heterogeneity-aware scheduling algorithm in a real operating system, (2) comparing it to dynamic algorithms and evaluating whether our algorithm's lack of ability to track changes is a serious drawback, (3) extending our signature-based framework to distinguish between a larger set of heterogeneous cores' features. In addition, the framework is theoretically extendable to take into account more complicated scenarios such as multithreaded applications or applications that are I/O bounded (rather than memory bounded). These extensions, however, are not trivial to make, and likely require a more powerful microarchitecture-indepenedent characterization apparatus. Therefore, they are outside of our immediate plans.

Other studies have concluded that heterogeneous multicore systems are able to achieve superior performance/energy ratio compared to similar homogeneous systems. However, this superior performance can only be realized when cores and applications running on them are assigned to match each other's properties. Therefore, any benefit is lost unless the operating system is heterogeneity-aware. This interdependence of the software and hardware means that hardware manufacturers have little incentive to create heterogeneous systems. This is a chicken-and-egg problem. By taking this opportunity to develop heterogeneity aware-designs before the hardware is available, we are able to facilitate and influence its development and adoption.

6. REFERENCES

[1] Intel® 64 and IA-32 Architectures Optimization Reference Manual. *Intel Corporation,* 2007

[2] Intel® 64 and IA-32 Architectures Software Developer's Manual. *Intel Corporation,* 2007

[3] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, and P. Husbands. The Landscape of Parallel Computing Research: A View From Berkeley. *UC Berkeley Technical Report UCB/EECS-2006-183,* 2006

[4] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proceedings of the Conference on Computing Frontiers,* 2006

[5] E. Berg and E. Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis. *In Proceedings of International Symposium on Performance Analysis of Systems And Software*, 2004

[6] S. Borkar. Thousand Core Chips—A Technology Perspective. In *Proceedings of the DAC,* 2007

[7] Alexandra Fedorova, D. Vengerov, and Daniel Doucette. Operating System Scheduling On Heterogeneous

Multicore Systems. In *Proceedings of the the PACT'07 Workshop on Operating System Support for Heterogeneous Multicore Architectures,* 2007

[8] K. Hoste and L. Eechhout. Microarchitecture-Independent Workload Characterization. *IEEE Micro Hot Tutorials*, 27(3):63-72, 2007

[9] R. Kumar, K. Farkas, N. Jouppi, R. Parthasarathy, and Dean 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,* 2003

[10] R. Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, N. Jouppi, and K. Farkas. Single-ISA Heterogeneous Multicore Architectures for Multithreaded Workload Performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture,* 2004

[11] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddy, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation,* 2005

[12] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems,* 2002

[13] Alan Jay Smith. Cache memories. *ACM Computing Surveys, 14(3):473–530*, September 1982

[14] Burton Smith. Many-Core Operating Systems. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), in conjunction with ISCA-34, Keynote Speech,* 2007