Trace Alignment Algorithms for Offline Workload Analysis of Heterogeneous Architectures

Muhammet Mustafa Ozdal
Intel Corporation
Hillsboro, OR 97124
mustafa.ozdal@intel.com

Aamer Jaleel
Intel Corporation
Hudson, MA
aamer.jaleel@intel.com

Paolo Narvaez
Intel Corporation
Hudson, MA
paolo.narvaez@intel.com

Steven Burns
Intel Corporation
Hillsboro, OR
steven.m.burns@intel.com

Ganapati Srinivasa
Intel Corporation
Hillsboro, OR
ganapati.srinivasa@intel.com

ABSTRACT

Heterogeneous architectures with single-ISA asymmetric cores have the potential to improve both the performance and energy efficiency of software execution by dynamically selecting the most appropriate core type to run each execution thread. In this paper, we propose a trace-based methodology to explore power and performance benefits of single-ISA heterogeneous core architectures. The basic idea is to collect multiple traces by running a workload on different homogeneous platforms, and to align these traces for offline analysis. For this, we propose a wavelet-based similarity metric, which captures both fine-grain and coarse-grain software phases across different traces. Then, we propose a scalable dynamic programming algorithm to optimize this metric to align the traces. Our experiments show that the runtime and energy values predicted by our offline methodology have good accuracy with respect to the real measurements from a prototype heterogeneous system. The proposed methodology can enable design space exploration of single-ISA heterogeneous multi-core systems using traces from off-the-shelf homogeneous systems.

1. INTRODUCTION

Single-ISA heterogeneous multi-core architectures have the potential to offer both high performance and low energy consumption by utilizing multiple asymmetric cores. Some commercial products in this category include Nvidia's Kal-El [11] and ARM's big.LITTLE [4] architectures, which combine high-performance ("big") cores with energy efficient ("small") cores on the same chip. While compute-intensive software phases can be run on a big (e.g. out-of-order) core to improve performance, the phases with inherently low instruction-level parallelism (ILP) can be run on a small (e.g. in-order) core to reduce the overall energy consumption.

The benefits of such heterogeneous architectures depend on the application characteristics, power-performance metrics, and the core selection algorithms utilized. It is possible that for certain applications, one core type is always the best choice, e.g. small cores for applications with very low ILP in power-restrained settings. However, in general, a combination of different core types may lead to the best results. For example, consider an application with nonuniform execution phases. For the best tradeoff between high performance and energy efficiency, the application can be assigned to a big core during performance intensive phases, and vice versa. Obviously, the scheduling algorithm that determines the core selection during runtime also affects how much of the potential benefits can be achieved in practice.

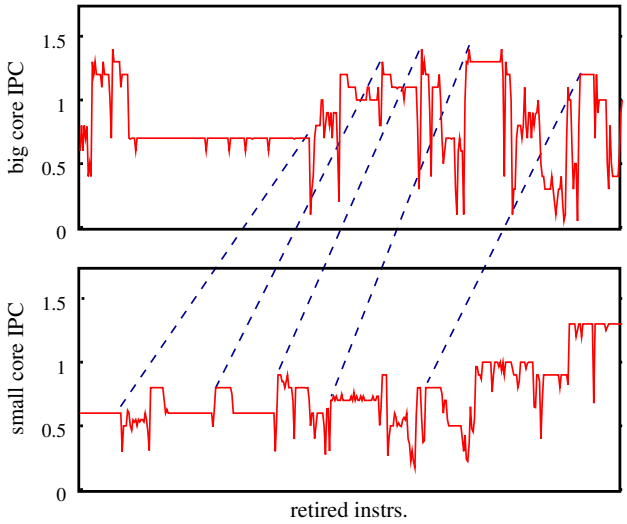

Figure 1: A snippet of two traces collected by running workload *gcc* (SPEC-INT) on "big" and "small" cores at different times. The instruction-per-cycle (IPC) values are plotted against the number of instructions retired. The dashed lines show the ideal alignment based on visual inspection of the IPC patterns.

The benefits of incorporating different core types in a system need to be studied for the target class of applications during architecture exploration phases. For this purpose, performance accurate simulators for different core types can be utilized. However, the main drawback is that the runtime requirements of the performance accurate simulators allow analysis of only very short (e.g. 1-10 seconds) snippets of applications. On the other hand, the benefits of heterogeneous systems are typically observed over longer durations as the applications go through different execution phases, each of which can be best suited for a certain core type. Furthermore, the selection of core type is often dependent on the thermal state of the system. With typical thermal time constants in the order of minutes, it can take hours of workload execution time to evaluate the average performance and energy usage of a heterogeneous platform. This would take several months (or years) of simulation time with the state-of-the-art detailed performance simulators.

An alternative approach is to run the workloads to be analyzed on existing off-the-shelf homogeneous systems and collect traces cor-

978-1-4799-1071-7/13/$31.00 ©2013 IEEE

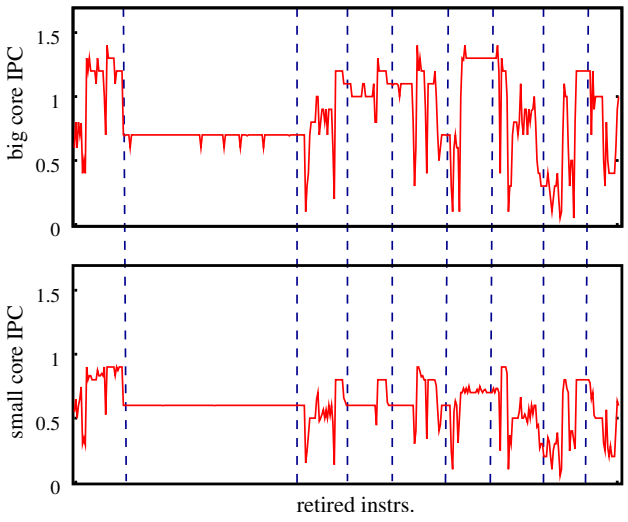

Figure 2: The two traces in Figure 1 are aligned using the algorithms proposed in this paper. The dashed lines show how the two traces were aligned based on the similarity of the IPC patterns.

responding to periodic power/performance event counter samples. These traces can be collected using kernel sampling mechanisms for each application. For example, assume that we want to analyze a serial workload W during the architecture exploration phase of a future heterogeneous system with two core types B (big) and S (small). For this purpose, one can run W on an off-the-shelf system with core type B, and collect a trace T^B, which contains the power/performance characteristics of B during execution of W. Afterwards, W can be run on another off-the-shelf system with core type S, and trace T^S can be collected in a similar way. Once the traces T^B and T^S are available, one can analyze the power and performance of W on a heterogeneous system (with core types B and S) by simulating a core scheduling policy. This can be done because the power/performance characteristics of each interval of W is stored in T^B and T^S separately. An offline scheduler can assign each interval of W to a core type based on the information in T^B and T^S, and the power/performance of W can be estimated on a future heterogeneous system.

This method implicitly assumes that traces T^B and T^S are aligned in instruction count. In other words, it assumes that instruction count N for T^B is the same as for T^S. Only this way, one can compare and evaluate the power/performance characteristics of a specific execution interval on cores B and S. However, in practice, traces T^B and T^S are never perfectly aligned. The main reason for this is the presence of background tasks and kernel processes during the execution of workload W. Due to the non-determinism of these processes, the number of instructions retired may differ slightly in different traces. Furthermore, since this error is cumulative, over time, T^B will drift away from T^S to the point where the instructions being executed around instruction count N on T^B are unrelated to the same neighborhood in T^S.

Figure 1 illustrates a snippet of two traces collected by running the *gcc* workload (from SPEC-INT benchmarks) on two core types. The first trace is from an Intel Core i7 ("big") core, while the second trace is from a defeatured ("small") core[1]. Although the full traces correspond to hundreds of seconds of execution, this figure illustrates a snippet of only a few seconds for better visibility. Both traces were collected with 4ms granularity using hardware counters, and they were aligned to each other based on the number of instructions retired (x axis). It is possible to observe the misalignment of the IPC patterns (y-axis) between the two traces. The ideal alignment of these snippets is shown with the set of dashed lines in the figure. This alignment can be inferred by visual inspection of the traces based on the IPC change patterns. Although such an alignment can be done manually for a small snippet, it is too time consuming to do this for a trace of size hundreds of times larger.

The objective of this paper is to automate the trace alignment process to enable offline analysis of execution traces collected from different cores. The difficulty of this problem is due to the following reasons:

- The misalignment in the number of retired instructions can be arbitrary because of the nondeterministic execution of the background tasks and kernel processes, which cannot be disabled during workload execution. In the example of Figure 1, the small core trace needs to be shifted towards right for better alignment. However, the exact shift value and the direction can be different at different points in the trace.

- The absolute and the relative IPC values are different when the same set of instructions are executed on different core types. In Figure 1, it is possible to observe some resemblance between the two traces in terms of how the IPC values change over time. However, the shape of the patterns are still significantly different, and automatically capturing the high-level resemblance and aligning the traces accordingly is not straightforward.

Figure 2 illustrates the result of our proposed algorithms on the two traces of Figure 1. Observe that the high-level similarity between the traces is captured almost perfectly. Furthermore, the traces are aligned in such a way that there is one-to-one correspondence between the intervals in the big and small core traces. Given such aligned traces, it is possible to compare the power/performance characteristics of different core types for every sampling interval of workload W.

The main contributions of this paper can be summarized as follows:

- For a given trace, we propose a wavelet-based model to define trace features corresponding to the IPC change patterns of different granularities. This model helps capture both coarse-grain and fine-grain software phases.

- Given multiple traces from different types of cores, we propose a similarity metric that uses the wavelet features defined for each trace. Intuitively, this metric captures the similarity between trace entries based on how IPC values change over time.

- We propose runtime improvements for the dynamic programming based algorithm for sequence matching. The proposed algorithms can match traces with more than hundred thousand entries within practical runtimes.

- In our experimental study, we show that the results of the proposed offline methodology correlate well with the real measurements from a heterogeneous system.

The proposed methodology can be used for design space exploration of heterogeneous multi-core systems by collecting traces from off-the-shelf homogeneous systems.

The rest of the paper is organized as follows. In Section 2, we provide a brief summary of the related work. Then, we formulate the trace alignment problem in Section 3. In Section 4.1, we propose a wavelet-based model to distinctly identify each trace interval based on software phases of different granularities. Based on this model, we define a similarity metric between two different traces in

[1]The defeaturing was done at hardware level to emulate small core power and performance. See Section 6.2 for details.

Section 4.2. Then, in Section 5, we propose an efficient dynamic-programming algorithm to align two traces such that the similarity metric is maximized. Our experimental results in Section 6 demonstrate the accuracy of our methodology with respect to a prototype system with heterogeneous cores.

2. RELATED WORK

Since the benefits of heterogeneous CMPs heavily rely on the workload-to-core mapping (i.e., scheduling policy), a significant amount of work has focused on novel scheduling policies to improve heterogeneous CMP performance. Specifically, existing proposals have investigated sampling-based scheduling [9], profile-based scheduling [2], model-based scheduling [3], or the use of specific performance-counters (e.g. cache misses) to guide scheduling [8].

The majority of prior studies on heterogeneous CMPs have been evaluated using performance simulators. Unfortunately, the excruciatingly slow speeds of the state-of-the-art performance simulators limits studies to a very short duration on the target system (e.g. 1-10 seconds) [12]. While performance simulator based studies are applicable for studying the system performance within very short time periods, a simulation based study is impractical for more realistic energy efficiency studies. Specifically, the typical thermal time constants require simulating several minutes of a target system, which may take months of simulation time with the detailed performance simulators.

On the other hand, only a few of the prior works performed evaluations by running workloads on real (prototype) heterogeneous systems. Specifically, [5, 8] used proprietary hardware mechanisms to "defeature" an existing out-of-order Intel Xeon® *(big)* core to emulate an in-order Intel Atom™ *(small)* core, e.g. by reducing the retired instruction count from four to one micro-op per cycle. This allowed them to run workloads on a system with 1 big core and 3 small cores, all on the same chip. Although such systems can be used to run realistic workloads, they are not freely available to the wider research community. Besides, it may be expensive to build such prototype systems, and kernel-level scheduling algorithms may be needed as in [8].

The methodology we propose is based on collecting separate execution traces from existing off-the-shelf cores, and then aligning the samples across multiple traces. Trace-based analysis have been widely used for homogeneous systems, and the need for trace alignment was discussed in [6, 10]. The motivation for trace alignment on homogeneous cores is to be able to collect tens or hundreds of metrics for a workload execution. Collecting all the metrics in a single run is not possible because of the limited number of hardware performance counters, and the potential overhead of monitoring more than a few metrics. So, the same workload can be run multiple times, each time collecting a different set of metrics. Due to the non-deterministic execution of kernel tasks, it was observed by both [6] and [10] that the traces collected were not aligned even if the same workload is executed on the same system. For this, trace alignment algorithms were proposed based on dynamic programming in [6] and [10]. Both of these algorithms are based on Dynamic Time Warping (DTW) [1], a technique used in speech recognition. The basic idea is to compute an ordered matching between individual trace entries such that the total IPC mismatch between aligned entries is minimized. The similarity between entries across multiple traces is modeled in a trivial way, because the IPC values are expected to be identical across multiple runs. Specifically, the dynamic programming objective in both [6] and [10] is to minimize:

$$DTWError = \sum_{k=1}^{|W|} |x_i - y_j|, \text{ where } w_k = (i,j) \qquad (1)$$

Here, the alignment is represented as a sequence W of pairs (i,j), where x_i and y_j are the IPC values of the aligned entries from two different traces. As can be seen, it is implicitly assumed that the IPC values are near-identical for the aligned entries.

On the other hand, when the traces collected are from different types of cores, such an assumption no longer holds. When the same sequence of instructions is executed on a small core and a big core, the performance differences can be significant. For example, for an execution phase with high instruction level parallelism (ILP), a superscalar (big) core can have significantly larger IPC values than a single-issue (small) core. Conversely, for a memory-bound phase, the difference between IPC values is likely to be smaller. So, a similarity metric based on IPC values alone is not enough to align traces from different types of cores. In this paper, we propose wavelet based models to capture both fine-grain and coarse-grain software phases. To the best of our knowledge, this is the first study that illustrates trace collection and trace alignment to enable arbitrary length energy efficiency studies on target heterogeneous platforms.

3. PROBLEM FORMULATION

Let trace T be defined as a sequence of periodic intervals, each with a set of values for specific power and performance event counters. In this paper, we focus on traces collected through kernel sampling mechanisms. In particular, the traces we collect are from a system that allows sampling counter values at every 4ms intervals. For each such interval, our traces contain the values corresponding to the number of instructions retired, stall cycles, sleep state residencies, average power consumption, etc. for each core in the system. Readers can refer to [7] for details on how to sample different performance counters for an Intel processor.

Assume that there are n intervals stored sequentially in trace T, and let $T[i]$ denote the i^{th} such interval, where $1 \leq i \leq n$. We can denote a sequence of intervals from (including) $T[i]$ to (excluding) $T[j]$ as $T[i,j)$, where $1 \leq i \leq j \leq n$. Obviously, $T[i,i)$ corresponds to an empty interval, and $T[i,i+1)$ corresponds to $T[i]$. Let $T[i,j).instrs$, $T[i,j).cycles$, and $T[i,j).IPC$ denote the number of instructions retired, the clock cycles spent, and the average number of instructions per cycle (IPC) in the sequence $T[i,j)$.

For simplicity of presentation, we will focus on matching only two traces. Let T_R be the reference trace collected by running workload W on core type R, and let T_M be the trace collected by running the same workload on core type M at a different time. The trace alignment problem is to match each interval $T_R[i]$ to a (possibly empty) sequence of intervals $T_M[j,k)$ such that the matched intervals have the maximum *similarity*. The similarity metric needs to be modeled properly to capture software phases, as illustrated in Figures 1 and 2. Our proposed model to capture the similarity between different intervals will be explained in more detail in Section 4.

For proper trace alignment, additional constraints need to be enforced on the matching problem. Consider any pair of indices i_1 and i_2 such that $1 \leq i_1 < i_2 \leq n$. Let $T_M[j_1,k_1)$ and $T_M[j_2,k_2)$ be the intervals matched to $T_R[i_1]$ and $T_R[i_2]$, respectively. The following constraints need to be satisfied for the matching problem:

1. Matching without overlaps: $[j_1,k_1) \cap [j_2,k_2) = \emptyset$.
2. Ordered matching: $j_1 < j_2$.
3. Continuous matching: If $i_2 = i_1 + 1$, then $j_2 = k_1$.

Figure 3 illustrates an example matching solution, where the constraints above have been satisfied.

It is possible to extend this formulation to handle the alignment of an arbitrary number of traces, each collected by running the same workload W on different types of cores or under different power settings[2]. One of these traces can be chosen as the reference trace

[2]For example, one can collect a different trace corresponding to each dynamic voltage frequency scaling (DVFS) configuration of each core.

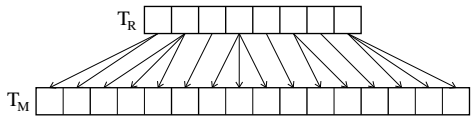

Figure 3: A matching solution between traces T_R and T_M.

T_R, and each of the remaining traces can be aligned with T_R one by one, leading to the alignment of all traces. An offline simulator using these aligned traces can then rapidly provide us with the overall performance/energy characteristics of a heterogeneous architecture and its core-scheduling algorithm running workload W.

The algorithms proposed in this paper are for single-threaded workloads. Furthermore, we focus on heterogeneous systems with infrequent task migrations (e.g. every few miliseconds). This way, the runtime overheads due to core migrations (e.g. cache warm-up penalties, etc.) are negligible compared to the overall execution times. This methodology can be extended in a future work to handle multiple threads and to incorporate migration overheads.

4. MODELING SIMILARITY BASED ON WAVELETS

We have formulated the trace alignment problem in Section 3 in terms of matching each interval $T_R[i]$ to a sequence of intervals $T_M[j,k)$. For this, we first need to define a similarity metric such that maximizing it during matching is expected to align the intervals corresponding to the same software phases. Visual inspection of Figure 2 shows that the actual IPC values of the aligned intervals can differ significantly because of the fact that these traces are collected from different types of cores. Hence, a simple similarity metric based on IPC values as in [6, 10] cannot be used here.

On the other hand, the IPC change patterns are similar between the two traces in Figure 2, because these traces correspond to the execution of the same workload. For example, if the workload involves a compute-intensive phase followed by a memory-bound phase, it is expected that the IPC values decrease in the corresponding trace intervals of both cores (but possibly with different amounts). Based on this intuition, we propose a trace feature model using wavelets in Section 4.1. Then, we propose a similarity metric for alignment in Section 4.2.

4.1 Wavelet-Based Feature Modeling

Wavelets are used to decompose mathematical functions hierarchically. In other words, they can describe a function in terms of a coarse overall shape, plus different levels of coarse-to-fine-grain details [14]. They have been used in many signal processing, computer graphics, and computer vision applications.

The continuous wavelet transform (CWT) of a function $f(t)$ can be obtained by scaling a *mother wavelet* (such as the Haar wavelet in Figure 4(a)) by different scaling factors and convolving them with $f(t)$. Unlike Fourier transform, CWT of a signal allows a time-frequency representation with both time and frequency localization. In other words, CWT of $f(t)$ helps analyze the spectral components at different time intervals of $f(t)$. In this section, we will make use of wavelet coefficients to model the trace features based on IPC change patterns of different frequencies.

Let us consider a given trace T as a function where $T[t]$ is the IPC value at interval t. Let $\psi_i[t]$ denote the Haar wavelet with scale $s = i$. We can compute the *wavelet coefficients* of T corresponding to $\psi_i[t]$ by computing the following convolution:

$$(T * \psi_i)[t] = \sum_{\tau=-\infty}^{\infty} T[\tau] \cdot \psi_i[t-\tau] \quad (2)$$

Figure 4 illustrates $\psi_1[\tau]$, $\psi_1[t-\tau]$, $\psi_2[t]$, and $\psi_2[t-\tau]$ in parts (a)-(d). Observe that $(T * \psi_1)[t] = T[t+1] - T[t]$, and $(T * \psi_2)[t] = (T[t+2] + T[t+1]) - (T[t] + T[t-1])$.

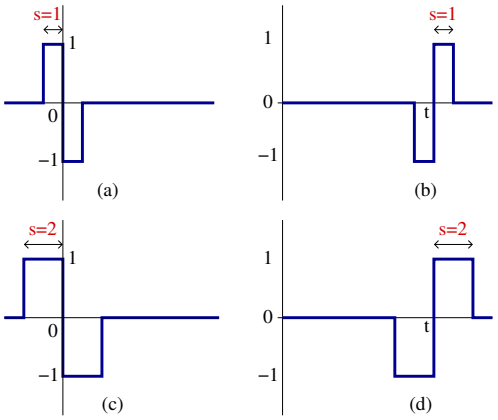

Figure 4: (a) The Haar wavelet $\psi_1(\tau)$. (b) The translated wavelet $\psi_1(t-\tau)$. (c) The scaled wavelet $\psi_2(\tau)$. (d) The scaled and translated wavelet $\psi_2(t-\tau)$.

In general, the wavelet coefficients for ψ_i are the difference values between the last i intervals and the next i intervals at each point t. Intuitively, if the scale value i is small, the wavelet coefficients correspond to the amplitudes of the high frequency components at different points, and vice versa. By computing the wavelet coefficients for different scale values, we can obtain a range of spectral components at each point t.

Since it is not computationally efficient to compute the wavelet coefficients for each ψ_i, we will focus on only a few sufficiently different scale values i to model the trace features. Specifically, let $W_T^f[t]$ denote the wavelet coefficient for $\psi_{(2^f)}$ at $T[t]$:

$$W_T^f[t] = (T * \psi_{(2^f)})[t] \quad (3)$$

By computing each $W_T^f[t]$ for $0 \leq f \leq f_{max}$, we can obtain $f_{max}+1$ different *features* at each point t, corresponding to high-frequency (small f) and low-frequency (large f) changes in the IPC values.

To normalize the features across different scale values, we use the *z-score* concept from statistics. The *z-score* (also known as the standard score) of a given sample x indicates by how many standard deviations x is above or below the mean of the population. We can compute the normalized wavelet coefficients (Z_T^f) using this definition as follows:

$$Z_T^f[t] = \frac{W_T^f[t] - \mu_T^f}{\sigma_T^f} \quad (4)$$

where μ_T^f and σ_T^f are the mean and the standard deviation of the sequence W_T^f, respectively.

In our implementation, we compute 6 feature sets Z_T^0 to Z_T^5 for each trace T. Figure 5 illustrates the Z_T^0 and Z_T^5 values for a snippet of the *gcc* workload on the big core. Observe that while Z_T^0 captures the fine-grain (high-frequency) changes in the IPC values, Z_T^5 captures the coarser-grain (lower-frequency) changes. Although not shown here due to page limitations, the plots of the corresponding Z_T^0 and Z_T^5 for the small core snippet also have very similar shapes to the ones in Figure 5.

4.2 Modeling Similarity Between Traces

As discussed earlier, the $Z_T^f[t]$ values (for $0 \leq f \leq f_{max}$) of a trace can capture software phases of different granularities. Let us now consider two traces obtained by running the same workload on different types of cores. Even though the actual IPC values can be significantly different, the software phases across different cores are

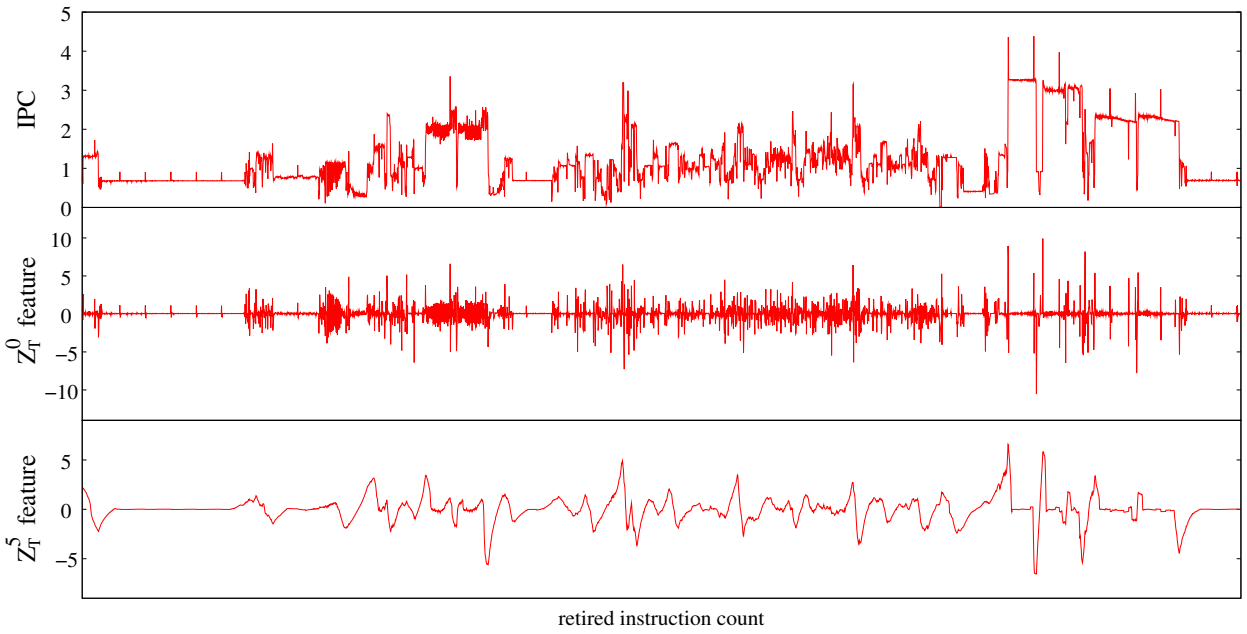

Figure 5: The IPC, Z_T^0, and Z_T^5 values are plotted for a 16-second snippet of the *gcc* workload (SPEC-INT) on the big core.

still expected to be *similar*. In this section, we propose a similarity metric between different core traces based on how the software phases match each other.

In statistics, *Pearson product-moment correlation coefficient* (PPMCC) [13] can be used to measure the correlation between two sequences X and Y, and is defined as follows:

$$PPMCC(X,Y) = \frac{1}{n-1} \sum_{(x_i,y_i)\in(X,Y)} \left(\frac{x_i - \mu_x}{\sigma_x}\right) \cdot \left(\frac{y_i - \mu_y}{\sigma_y}\right) \tag{5}$$

where μ_x, σ_x, μ_y, and σ_y correspond to the mean and the standard deviation of the sequences X and Y, respectively. Observe that the summation is over the products of the *z-scores* (as described in Section 4.1) of the individual entries of X and Y. Note that a larger $PPMCC(X,Y)$ value indicates higher correlation between X and Y, and vice versa. Intuitively, if the two signals corresponding to X and Y have matching peaks and valleys, then this will lead to a higher $PPMCC(X,Y)$ value.

In Section 4.1, we showed how the wavelet coefficients of different scales correspond to software phases of different granularities. Hence, aligning the software phases of two traces T_R and T_M can be achieved by maximizing the correlation between their corresponding wavelet coefficients. Based on this, the first alignment objective can be formulated as follows:

$$maximize \quad \sum_{f=0}^{f_{max}} PPMCC(W_R^f, W_m^f) \tag{6}$$

where W_R^f and W_M^f are the wavelet coefficients of T_R and T_M, respectively, as defined in Equation 3. Substituting (4), (5), and removing the constant term, we can rewrite (6) as:

$$maximize \quad \sum_{i=1}^{n} \sum_{f=0}^{f_{max}} Z_R^f[i] \cdot Z_M^f[j,k] \tag{7}$$

where each $T_R[i]$ is aligned to $T_M[j,k)$ as formulated in Section 3.

Since our focus is on heterogeneous systems with cores of same ISA, our second alignment objective is to minimize the mismatch in the number of instructions retired[3], which is denoted as r_{mm}, and defined as:

$$r_{mm} = \frac{|T_R[i].instrs - T_M[j,k).instrs|}{T_R[i].instrs} \tag{8}$$

We can incorporate r_{mm} into our initial objective (7) to obtain the final alignment objective function:

$$maximize \quad \sum_{i=1}^{n} sim\left(T_R[i], T_M[j,k)\right) \tag{9}$$

where:

$$sim\left(T_R[i], T_M[j,k)\right) = (1 - r_{mm}) \sum_{f=0}^{f_{max}} Z_R^f[i] \cdot Z_M^f[j,k) \tag{10}$$

In summary, Equation 10 defines the similarity between intervals $T_R[i]$ and $T_M[j,k)$ based on 1) the retired instruction counts, and 2) the correlation between the normalized wavelet coefficients.

5. TRACE ALIGNMENT ALGORITHM

In this section, we propose an algorithm to solve the trace alignment problem as formulated in Section 3 to maximize the similarity function defined in Equation (10). The basic dynamic programming (DP) formulation is similar to the DTW technique used in [6] and [10], and is explained in Section 5.1. This basic algorithm has high time complexity with respect to the size of the traces. Hence, it is not scalable enough to handle very long execution traces. We propose improvements to this DP formulation in Section 5.2 to reduce its asymptotic complexity.

5.1 Dynamic Programming Formulation

Let the reference trace be T_R, with size n, and let the trace to be matched be T_M with size m. As before, assume $T[i,j]$ and $T[i,j)$

[3]As discussed before, occasional mismatches in the retired instruction counts are unavoidable because of the non-deterministic execution of kernel tasks, etc. However, such mismatches should be minimal between aligned traces.

denote the sequence of intervals $[i, j]$ and $[i, j)$ in trace T, respectively[4].

The DP formulation is based on the following optimal substructure property.

Property 5.1. *Consider the problem of matching two traces $T_R[1, n]$ and $T_M[1, m]$. Let A be an optimal matching solution for this problem, where the last element of T_R, $T_R[n]$, is matched to $T_M[j_n, k_n]$. Let $A' \in A$ be the matching solution corresponding to intervals $T_R[1, n-1]$. It must be the case that A' is an optimal solution for the subproblem of matching $T_R[1, n-1]$ to $T_M[1, j_n)$.*

Let $s[i, k]$ denote the optimal score for the problem of matching $T_R[1, i]$ to $T_M[1, k)$. The following equation must hold:

$$s[i,k] = \max_{1 \le j \le k} \{s[i-1, j] + \textit{similarity}(T_R[i], T_M[j,k))\}$$

where the similarity function is as defined in Equation (10).

According to Property 5.1, the problem of matching $T_R[1, i]$ to $T_M[1, k]$ can be solved by choosing the j value that maximizes the sum of 1) the optimal cost of the remaining subproblem of matching $T_R[1, i-1]$ to $T_M[1, j)$, and 2) the similarity score of matching $T_R[i]$ to $T_M[j, k)$. Based on this, a dynamic programming algorithm can be formulated to compute the partial solutions iteratively in a bottom up fashion, and this algorithm is guaranteed to compute the optimal alignment solution for traces $T_R[1, n]$ and $T_M[1, m)$.

5.2 Runtime Improvements

The asymptotic runtime complexity of the basic DP algorithm proposed in Section 5.1 is $\theta(n \cdot m^2)$, which is too high for long traces. In this section, we propose heuristic improvements to reduce this runtime to $\theta(n+m)$ while maintaining the solution quality. The runtime improvements will be based on two observations. Assume that the interval $T_R[i]$ is matched to $T_M[j, k)$ in an optimal matching solution for the following.

Observation 5.1. *It is expected that the workload* phase *executed during $T_R[i]$ is the same phase executed during $T_M[j, k)$, because they are matched to each other in the optimal solution. Since the cores R and M are assumed to have the same instruction set architecture, the number of instructions retired during $T_R[i]$ and $T_M[j, k)$ are also expected to be similar.*

Observation 5.2. *Due to Observation 5.1, the number of instructions executed in $T_R[1, i-1]$ is expected to be reasonably close to the number of instructions executed in $T_M[1, j)$.*

According to Observation 5.1, we only need to consider the matching solutions where the instruction counts of the matched intervals are reasonably close. In other words, we can define the following constraint for a valid matching between $T_R[i]$ and $T_M[j, k)$:

$$r_1 \le \frac{T_R[i].instrs}{T_M[j,k).instrs} \le r_2 \qquad (11)$$

where r_1 and r_2 are predefined tolerance values for instruction count mismatches. In our implementation, we use $r_1 = 0.5$ and $r_2 = 1.5$ to allow up to 50% mismatches in the instruction counts of the matched intervals. Note that despite such a relaxed constraint, our similarity function in Equation (10) still penalizes the mismatches in the instruction counts.

According to Observation 5.2, we can compute the $s[i, k]$ scores only for the intervals $T_R[1, i]$ and $T_M[1, k]$ that have reasonably close instruction counts. For a given $T_R[i]$, let k_i be the index chosen to minimize the expression $|T_R[1, i].instrs - T_M[1, k_i].instrs|$.

[4]The half-open intervals are used to enable matching an interval $T_R[i]$ to an empty interval $T_M[k, k)$.

```
ALIGN-TRACES (T_R[1,n], T_M[1,m))
for i ← 1 to n do
  for k ← (k_i − ε) to (k_i + ε) do
    s[i,k] ← 0
    for j ← (k − δ) to k do
      if s[i−1,j] + similarity(T_R[i], T_M[j,k)) > s[i,k]
        s[i,k] ← s[i−1,j] + similarity(T_R[i], T_M[j,k))
        parent[i,k] ← j
construct the solution by backtracing from parent[n,m]
```

Figure 6: The proposed dynamic programming algorithm

In other words, if we were aligning traces only based on the total retired instruction counts, $T_R[i]$ would be matched[5] to $T_M[k_i]$. Based on Observation 5.2, we can compute the $s[i, k]$ scores only for the k values where $k \in [k_i - \epsilon, k_i + \epsilon]$, where ϵ is a predefined tolerance value. The value of ϵ should be chosen such that the misalignment due to the accumulated noise (i.e. the additional instructions from background kernel tasks, etc.) is expected to be less than ϵ. In our implementation, we have set $\epsilon = 1000$ for all our experiments, without fine tuning. For very long traces, our implementation also allows periodically backtracing from the partial solutions to reset the accumulated noise so that it never goes above the ϵ value chosen. Further details about our implementation are omitted due to page limitations, but will be published in an extended version in the future.

The pseudo code of our proposed algorithm is shown in Figure 6. In each iteration of this algorithm, we evaluate the score of matching interval $T_R[i]$ to $T_M[j, k)$. Due to constraint (11), the size of $T_M[j, k)$ must be bounded in terms of the number of instructions retired. Hence, the lower bound for j is set to be $k - \delta$, where δ is computed in every iteration to satisfy constraint (11). Since ϵ and δ values are bounded by a constant, the runtime complexity of this algorithm is $\theta(n + m)$.

6. EXPERIMENTAL RESULTS

6.1 Accuracy of the Alignment Algorithm

In the first set of experiments, our objective is to evaluate the accuracy of the proposed trace alignment algorithm. For this, we make use of proprietary performance accurate CPU simulators corresponding to a big core and a small core, and collect traces for snippets of SPEC-INT benchmarks. During each simulation, the exact sequence of instructions executed for the given workload is known (in contrast to the execution on a real system, where there are non-deterministic background tasks). So, the exact alignment information of the small and big core traces is available ahead of time. However, running these simulations is expensive, and we cannot collect long enough traces to demonstrate the benefits of heterogeneity. In this section, we will use these traces to evaluate the accuracy of our alignment algorithms.

Let performance scalability at time t for a particular task be defined as follows:

$$scalability = \frac{IPC_{big}}{IPC_{small}} \qquad (12)$$

where IPC_{big} and IPC_{small} denote the average IPC achieved if the same sequence of instructions are run on the big core and small core, respectively.

Our experimental methodology is as follows. We first collect big core trace T_R and small core trace T_M for each benchmark through detailed performance simulations. Then, for each interval $T_R[i]$, we compute the performance scalability as defined in Equation 12. After that, we artificially add noise to T_M (see below for details) to

[5]It is straightforward to show that for all i values, $1 \le i \le n$, the entire set of corresponding k_i values can be computed in $\theta(n + m)$ time as a preprocessing step.

Table 1: Experimental results showing the robustness of the proposed alignment algorithm

benchmark	NOISE=1%				NOISE=5%				NOISE=10%			
	Proposed Algorithm		Instr Cnt Based		Proposed Algorithm		Instr Cnt Based		Proposed Algorithm		Instr Cnt Based	
	80% accr.	avg. error	80% accr.	avg. error	80% accr.	avg. error	80% accr.	avg. error	80% accr.	avg. error	80% accr.	avg. error
astar_B	100%	1%	97%	8%	100%	4%	88%	23%	97%	8%	75%	39%
bwaves_b	100%	1%	70%	32%	99%	5%	65%	48%	92%	9%	61%	49%
bzip2_c	92%	5%	81%	11%	89%	8%	61%	25%	81%	13%	44%	36%
calculix_h	98%	2%	86%	12%	98%	6%	77%	16%	88%	11%	71%	19%
gamess_c	100%	1%	100%	2%	100%	4%	100%	4%	95%	8%	94%	8%
gcc_166	100%	1%	70%	30%	99%	5%	60%	38%	92%	9%	50%	38%
gemsFDTD_r	99%	1%	78%	27%	98%	5%	52%	49%	92%	9%	36%	57%
gobmk_s	99%	1%	92%	8%	99%	5%	92%	10%	91%	9%	87%	12%
h264ref_f	100%	2%	80%	12%	99%	5%	72%	14%	95%	8%	71%	15%
hmmer_n	97%	3%	69%	17%	95%	7%	48%	28%	86%	11%	46%	30%
mcf_r	100%	1%	97%	6%	100%	4%	95%	8%	95%	8%	86%	11%
milc_s	100%	1%	48%	32%	99%	5%	40%	49%	94%	9%	35%	67%
namd_n	100%	1%	100%	3%	100%	4%	100%	6%	94%	8%	93%	9%
omnetpp_o	100%	1%	92%	8%	100%	4%	92%	9%	94%	8%	85%	11%
perlbench_c	100%	1%	73%	14%	100%	4%	57%	21%	97%	7%	56%	21%
sjeng_r	100%	1%	100%	1%	100%	5%	100%	4%	93%	8%	96%	8%
soplex_p	93%	6%	82%	13%	91%	8%	80%	14%	87%	11%	74%	15%
wrf_r	99%	2%	59%	34%	96%	6%	41%	61%	86%	11%	32%	92%
xalancbmk_r	100%	1%	64%	18%	100%	4%	54%	23%	95%	8%	53%	23%
zeusmp_z	99%	2%	79%	16%	98%	6%	48%	30%	93%	9%	51%	33%
Avg	99%	2%	81%	15%	98%	5%	71%	24%	92%	9%	65%	30%

obtain the noisy trace T'_M. Using our proposed algorithms, we align T_R and T'_M for each benchmark, and compute the performance scalability for each interval $T_R[i]$ based on the alignment results. By comparing these scalability values with the original ones (before the addition of noise), we can evaluate how well our alignment algorithms perform in the presence of noise.

Our alignment algorithms were implemented in C++, and run on a 3GHz Xeon system. For all benchmarks in this experiment, the alignment execution took less than 4 seconds, which is significantly less than the runtime of the performance simulator.

Table 1 shows the results of this experiment. Here, we have stress-tested our algorithms by adding different amounts of noise to T_M. Specifically, "NOISE = X%" in this table means that the random noise added to each interval is from a Gaussian distribution with mean $X\%$ and standard deviation $2X\%$. In this table, we compare the results of our proposed algorithm with a straightforward alignment technique using retired instruction counts only.

The accuracy values in Table 1 are reported in terms of two values: 1) 80% accuracy, which is the percentage of the intervals with errors less than 20%, and 2) average error. For example, when 10% noise is added to benchmark *astar_B*, 97% of the intervals were predicted with less than 20% error, and the average error was 8%. The results in this table show that as the noise ratio increases, we cannot rely on only instruction counts for alignment. On the other hand, even for large noise ratios, our proposed alignment algorithms can still match the traces with reasonably high accuracy.

6.2 Accuracy of the Offline Methodology

In the second set of experiments, our purpose is to show that the results of our offline methodology correlates well with the actual real-time measurements collected by running the same workloads on a system with heterogeneous cores.

Our experiments have been performed on a prototype system that emulates a heterogeneous architecture with small and big cores. This system contains 4 Intel i7 cores, each of which can be *defeatured* using proprietary hardware mechanisms to emulate small core performance. A similar evaluation platform was used by the authors of [5, 8], and the readers can refer to them for further details. In the rest of this section, we will use the terms *big core* and *small core* to refer to an original Intel i7 core and a defeatured core, respectively.

A modified Linux kernel is run on this system to enable scheduling of tasks to heterogeneous cores. The prototype kernel scheduling algorithm estimates the performance scalability of the running process at every 4ms time intervals, and assigns the process to a big core if scalability is above a certain threshold (which is set to 2.0 in our experiments). Otherwise, it assigns it to a small core. The scalability prediction is done during runtime based on sampling IPC values periodically on small and big cores and training dynamic prediction models. The details of the prediction and scheduling algorithms are omitted due to page limitations.

In our experiments, we have configured the heterogenous system to consist of 1 big and 3 small cores. On this system, we have executed single-threaded SPEC-INT benchmarks, and measured the energy and runtime for each benchmark execution. In these runs, we have relied on the kernel scheduling algorithm (described above) to dynamically choose the appropriate core type at periodic time intervals and migrate the process between cores as needed. The results of these experiments are reported in Table 2, under columns titled *"online measurements"*. The energy values reported in this table are normalized with respect to the workload with the lowest energy.

To evaluate the accuracy of our offline methodology, we first configured the system as 4 big cores, and executed the same benchmarks to collect *big core traces*. Then, we reconfigured the system as 4 small cores, and repeated the runs to collect *small core traces*. Figure 1 shows a small snippet of the big core and small core traces for the *gcc* benchmark. After that, we aligned these traces using the algorithms proposed in this paper. The aligned traces for the same snippet are shown in Figure 2. A different snippet of the *gcc* benchmark is shown in Figure 7 with more details. Observe that both the high-frequency and low-frequency IPC changes are aligned almost perfectly between the small and big core traces using the algorithms proposed.

The runtime spent to align each benchmark is shown under the second column of Table 2. Observe that the alignment runtimes are less than the workload execution runtimes for all but one benchmark. This shows that the proposed methodology is scalable enough to be used to align very long traces.

To evaluate the accuracy of the offline analysis using aligned traces, we have implemented the same kernel scheduling policy (described above) for offline analysis. We simulated this scheduling policy using the aligned big core and small core traces. The results of this methodology are listed under column *"offline with our alignment algorithm"* of Table 2. Observe that the runtime and energy measurements of the online runs are very close to the results ob-

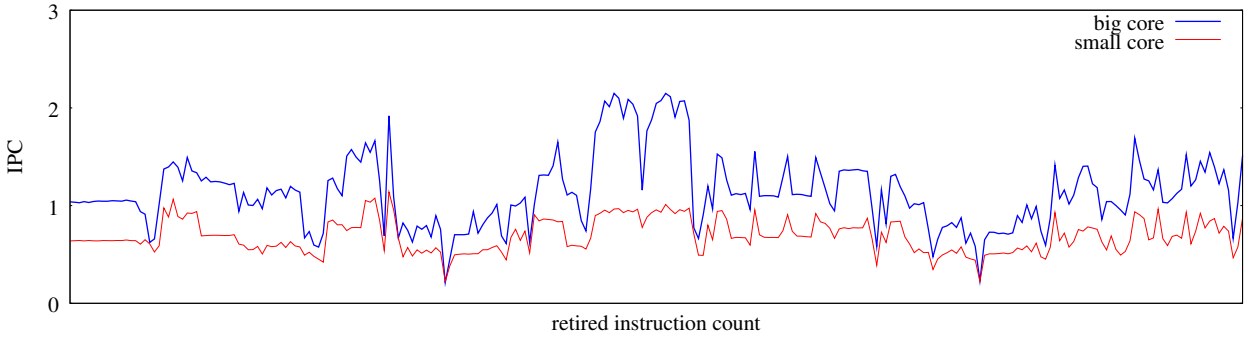

Figure 7: The alignment result of two traces for a snippet from the gcc benchmark

Table 2: Experimental results demonstrating the accuracy of the offline methodology

Benchmark	Alignment runtime (sec)	Online Measurements		Offline with our Algorithm				Offline with Instr. Cnt. Based Alignment			
		runtime (sec)	energy (norm)	runtime (sec)	energy (norm)	runtime (sec)	energy (norm)	runtime (sec)	energy (norm)	runtime (sec)	energy (norm)
gcc	492	876	46	888	49	1%	6%	666	53	24%	16%
perlbench	675	1170	74	1171	79	0%	7%	947	83	19%	13%
bzip2	771	1637	80	1714	80	5%	0%	1323	95	19%	19%
h264ref	898	1498	136	1590	137	6%	1%	1476	143	1%	5%
omnetpp	299	649	29	656	30	1%	1%	655	30	1%	1%
mcf	265	582	27	570	29	2%	7%	555	30	5%	9%
gobmk	617	1655	72	1651	74	0%	3%	1467	81	11%	12%
hmmer	470	260	62	255	62	2%	1%	273	62	5%	1%
sjeng	724	1845	76	1858	78	1%	2%	1865	79	1%	5%
astar	411	917	49	919	50	0%	2%	898	51	2%	4%
Avg						2%	3%			9%	8%

tained through the proposed offline methodology. This shows that accurate power/performance estimations of heterogeneous systems can be achieved by aligning traces collected from different homogeneous systems. This methodology also allows exploring different scheduling policies on the aligned traces without kernel-level implementations on a heterogeneous system.

For comparison purposes, we repeated this experiment for the traces aligned using instruction counts only, the results of which are listed under the last 4 columns of Table 2. Observe that simplistic trace alignment can lead to large errors (up to 24%) especially for benchmarks with non-uniform execution phases, such as *gcc, perlbench, bzip2,* and *gobmk*. On the other hand, aligning traces using the proposed algorithms leads to much more accurate results.

7. CONCLUSIONS

In this paper, we have proposed a trace alignment algorithm for offline workload analysis of heterogeneous systems. Our experiments demonstrate the accuracy of this methodology with respect to the measurements taken from actual executions on a heterogeneous system and performance accurate proprietary CPU simulators. The proposed methodology can be used for exploration of heterogeneous systems by collecting traces from off-the-shelf homogeneous systems. It can also be used for evaluating different scheduling policies without implementing them in a kernel and running on a real heterogeneous system.

8. REFERENCES

[1] D. J. Berndt and J. Clifford. Using dynamic time warping to find patterns in time series. In *Working Notes of the Knowledge Discovery in Databases Workshop*, pages 359–370, 1994.

[2] J. Chen and L. K. John. Efficient program scheduling for heterogeneous multi-core processors. In *DAC*, pages 927–930, 2009.

[3] K. V. Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *ISCA*, pages 213–224, 2012.

[4] P. Greenhalgh. Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7: Improving energy efficient in high-performance mobile platforms, 2011.

[5] V. Gupta and P. Brett et. al. Extending the dynamic power range of client devices using heterogeneous multicore processors. In *SHAW*, 2012.

[6] M. Hauswirth, A. Diwan, P. F. Sweeney, and M. C. Mozer. Automating vertical profiling. In *ACM SIGPLAN Conf. on object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 281–296, 2005.

[7] Intel Corporation. *Chapter 18: Performance Monitoring, Intel 64 and IA-32 Architectures Software Developer's Manual*. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[8] D. Koufay, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Eurosys*, pages 125–138, 2010.

[9] R. Kumar, K. I. Farkas, P. Jouppi, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO*, pages 81–92, 2003.

[10] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Aligning traces for performance evaluation. In *IEEE IPDPS*, 2006.

[11] Nvidia. Variable SMP - A multicore CPU architecture for low power and high performance, 2011.

[12] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer. Hasim: FPGA-based high-detail multicore simulation using time-division multiplexing. In *HPCA*, 2011.

[13] J. L. Rodgers and W. A. Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42:59–66, 1988.

[14] E. J. Stollnitz, T. D. DeRose, and D. H. Salesin. Wavelets for computer graphics: A primer, part 1. *IEEE Computer Graphics and Applications*, 15:76–84, 1995.