A Mathematical Model for Accurately Balancing Co-Phase Effects in Simulated Multithreaded Systems

Joshua L. Kihm, Tipp Moseley, and Daniel A. Connors
University of Colorado at Boulder
Department of Electrical and Computer Engineering
UCB 425, Boulder, CO, 80309
{kihm, moseleyt, dconnors}@colorado.edu

Abstract

As various types of multithreaded architectures can increase resource utilization and throughput of modern processors, there exists a very large design space which must be explored to effectively characterize these systems. Since these systems execute instructions from multiple concurrent application threads, understanding the interactions between co-scheduled threads is critical to evaluating potential designs. Unfortunately, understanding the interaction of co-scheduled threads requires complex and computationally intensive simulation. Although methods for accelerating architecture simulation are effective for applications demonstrating phase behavior over time in single threaded architectures, multithreaded systems exhibit even more widely variant behavior as different phases of each thread interact. Overall, there are many simulation technology issues related to phase behavior that must be investigated to effectively explore future multithreaded architectures. This paper explores how differences in multithreaded phase interactions and their relative frequencies impact overall performance evaluation. Further, a mathematical model is presented that balances the effects of inter-thread phase interactions in simulation to more accurately reflect the likely behaviors encountered in a real system.

1. Introduction

Various types of multithreading, including Simultaneous Multithreading (SMT) [4], Fine-Grained Multithreading [1], Coarse-Grained Multithreading [17], and Chip Multiprocessors (CMP) [14], have been proposed and implemented. The common theme among all multithreading techniques is that instructions from multiple threads (typically from independent programs) are executed within the same small time interval. Co-execution masks long latency events such as complex ALU functions, branch mispredictions, and accesses to lower levels of the memory hierarchy by executing instructions from other threads. The common mechanism of these systems is that some processor resources are shared between threads. This sharing ranges from CMP which can share low-level caches to SMT where essentially every microarchitecture resource is shared. The downside of any type of multithreading is that the collective set of requests can overwhelm the capacity of certain shared resources, causing interference between the threads. This interference heavily influences the performance of multithreaded systems [13], complicating the evaluation of multithreaded architectures.

The design exploration space for future multithreaded architectures is vast and it is difficult to quickly identify critical and optimal design decisions. Candidate designs are first evaluated by constructing simulators; unfortunately, simulation of real workloads is a major bottleneck in the design process and inaccurate simulation models have been shown to be multiple degrees from the quality of the final design [2]. In general, cycle accurate simulation of a complex, modern processor entails, at a minimum, a several thousand-fold slowdown over hardware. To explore the space of microprocessor design various simulation techniques and models have emerged. A commonly used solution to this problem is to exploit program phase behavior [3, 6, 10, 12, 8] to eliminate unnecessary simulation time by finding key representative phases within applications. Almost all programs can, to some degree, be divided into regions of common behavior called phases. Phases are differentiated based on either code usage [3, 8, 11] or performance data [15]. For the purpose of this paper, a program is divided into equally sized time slices called *periods*. The set of periods with similar behavior is a *phase*. Finally, a set of consecutive periods with the same phase is an execution *interval*. These concepts are illustrated in Figure 1. In a multithreaded system, behavior is dictated in large part by the phases of each of the co-scheduled threads. The

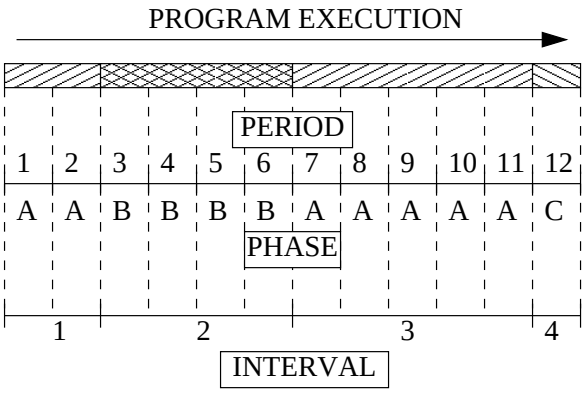

Figure 1. Program execution (represented in the first line)is broken into time slices called *periods*(the second line). Many periods demonstrate similar behavior called a *phase* (labeled on the third line). A set of consecutive periods in the same phase are called a *interval* (the last line)

resulting combination of phases from each thread in a multithreaded environment is termed a *co-phase* [16].

Traditionally, multithreaded systems are simulated by starting each thread together and allowing them to run together until one completes. The fundamental problem with this approach is that the combination of phases that interact between programs will vary in a real system depending on operating system scheduling. For multithreaded simulation, there are many similar instances in which real operating conditions and simulated operating constraints diverge, all of which can impair design decisions.

For instance, it is very unlikely that two programs of more than trivial length will remain co-scheduled for the duration of their execution. Other processes, including those from the OS, will likely use some amount of the processor's time. When any one of the target applications is not resident on the processor, the other threads, which are still resident, will advance and possibly change phase. Even if there is no phase change, the time until the next phase switch will be reduced. These effects lead to two major discrepancies to the single simulation approach. First, many co-phases may not be observed in one simulated run that may well occur in a hardware. These co-phases may exhibit unique and important behaviors that are simply neglected because they are not observed. Second, the balance between co-phases will be heavily skewed toward the simulated run which may emphasize a co-phases which, on average, are very uncommon and under-represent those which are important.

Although program phase has been recognized as a major factor in multithreaded execution, the combination of phases that interact between programs will vary in a real system has and not been fully explored. With this in mind, this paper addresses several missing elements of the characterization of the interaction between threads. This paper introduces a model which determines how different co-phase interactions should be weighted in a way that is representative of the amount it would occur in an OS scheduling environment.

The remainder of this paper is organized as follows. Section 2 supplies motivation for this research including data on the effects of start time offset in a common SMT processor, the Intel Pentium-4 with Hyper-threading. Section 3 walks through a simple, illustrative example with two threads from the *Spec2000* suite. Section 4 explains our multithreading model for two threads and and explains extensions to more than two threads, and Section 5 concludes and outlines future work.

2. Motivation

Recognizing program phase has important ramifications for simulation. Essentially, identification of repeating and representative phases makes it possible to simulate very small portions of a program's execution and then predict overall performance from those samples with high accuracy [11]. This offers advantages over random or statistical sampling where brief periods of execution are used to extrapolate overall behavior [18] in that only unique behaviors need to be simulated, requiring less simulation time [19]. This idea can be extended to simulation of multithreaded systems. The complication is that each combination of phases, or co-phase, can have unique interference. The number of co-phases is the product of the number of phases from each threads. As a result the total number of co-phases will grow exponentially with the number of threads. Faced with this, it is tempting (and probably necessary in many cases) to limit the number of co-phases that are simulated. However, it is important that as many co-phases as possible are tested. Just as program behavior can vary greatly between phases, the behavior in a multithreaded system will vary greatly between co-phases. Therefore, overall observed system behavior will be dependent on the combination of co-phases that are experienced.

To test the effects of co-phase mixes, several experiments were performed on a real system with support for multithreading and run-time performance monitoring. Five of the benchmarks from the *Spec2000* suite were run on an Intel Pentium-4 Northwood processor with Hyper-threading (a version of SMT) [5]. The benchmarks were chosen based on their long run-times, which allows a longer start time offsets to be tested. Additionally, the benchmarks were chosen to mix memory intensive with computation and control limited benchmarks and integer and floating point benchmarks. Each possible pairing of the benchmarks was run

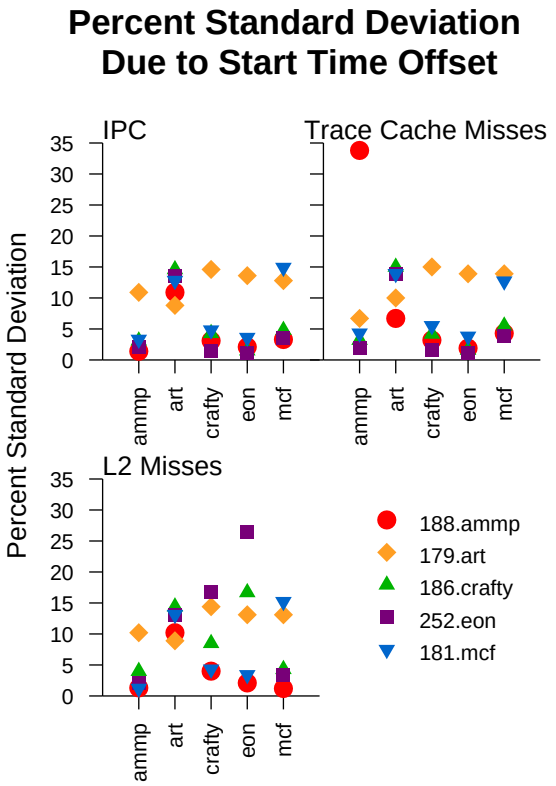

Figure 2. Standard deviation in multithreaded IPC, trace cache misses, and Level-2 cache misses due to start time offset for various *Spec2000* benchmark pairings.

with offsets in start time from negative to positive one hundred seconds, at ten second second intervals (-100, -90, -80, ..., 80, 90, 100), for a total of 21 tests for each pairing. The offset was used to produce a different combination of co-phases in each test. Using the performance counters on the Pentium-4, the instructions per cycle (IPC) and the rates of cache misses in the trace cache and the unified, level-2 cache were determined for the periods in which the threads were co-scheduled. The percent standard deviation between tests for each metric is presented in Figure 2. Each benchmark is represented by a column of the graph and a symbol. The first benchmark in the pairing is represented by the column of the graph, and the second by the symbol. Many pairings demonstrate deviation above ten percent in each of the categories, meaning that any single run of the pairings at a given offset would likely be significantly different from a run at another offset. The benchmark *179.art* is a particularly striking example of this behavior with high variance in all categories for all pairings. Generally, this means that when *179.art* is run with any of the selected applications, a selected time of offsetting the two applications can result in potential differences of 10% IPC, trace cache misses, and L2 misses when compared to a different offset. Overall, these results have significant implication on the techniques used to model and simulate multithreaded architectures.

In order to do divide the programs into phases, various performance counters on the Pentium-4 were used to sample performance at each scheduling interval (approximately 100ms) in a modified Linux 2.6.5 environment. Several runs are made on each benchmark and different performance counters from separate runs are combined into a single vector. The vectors from each sample are clustered using a k-means algorithm. Although more precise methods have been developed for phase analysis in hardware ([8]), using performance counters is sufficient for our purposes as only coarse-grained behaviors are needed for this work.

In addition to which co-phases occur, the time spent in each co-phase is also a product of the offset, which is illustrated in Figure 3. Using some of the data from Figure 2, these graphs illustrate the co-phase behavior of the benchmarks when they are paired with *252.eon*. Each row of graphs represents a different benchmark paired with *eon* and each column represents a different offset between the start times of the benchmarks. In each graph, the x-axis represents the phase of *eon* and the y-axis represents the phase of the other benchmark. The color or shade of each point on the graph represents the number of cycles spent in each co-phase with brighter colors indicating more cycles. The columns of graphs illustrate this behavior for offsets of negative one hundred, negative fifty, zero, fifty, and one hundred seconds. The variation between the graphs illustrates that co-phase behavior varies with offset. The difference due to offset is more quantitatively illustrated in Figure 4. The difference between two consecutive offsets is found by determining the percentage of execution that that occurs in each co-phase, then summing the difference between runs in these percentages across all co-phases (the number is divided by two to give a percentage). The graph demonstrates that for the benchmarks tested, a change in offset of fifty seconds results in at least a 10% difference in co-phase mix, and on average over a 20% difference.

In [16], it is recognized that when threads are run together, only a small number of the possible co-phases actually occur, especially for multithreaded systems with many threads. However, in any real world system, it is very unlikely that two threads will be started at the same time or even that the start time offset will be consistent between runs. Additionally, because other programs, including the OS, are competing for processor time, it is very unlikely that threads will be co-scheduled for the entire duration of their execution. This will also affect the co-phase mix. This means that characterizing a system based on starting the

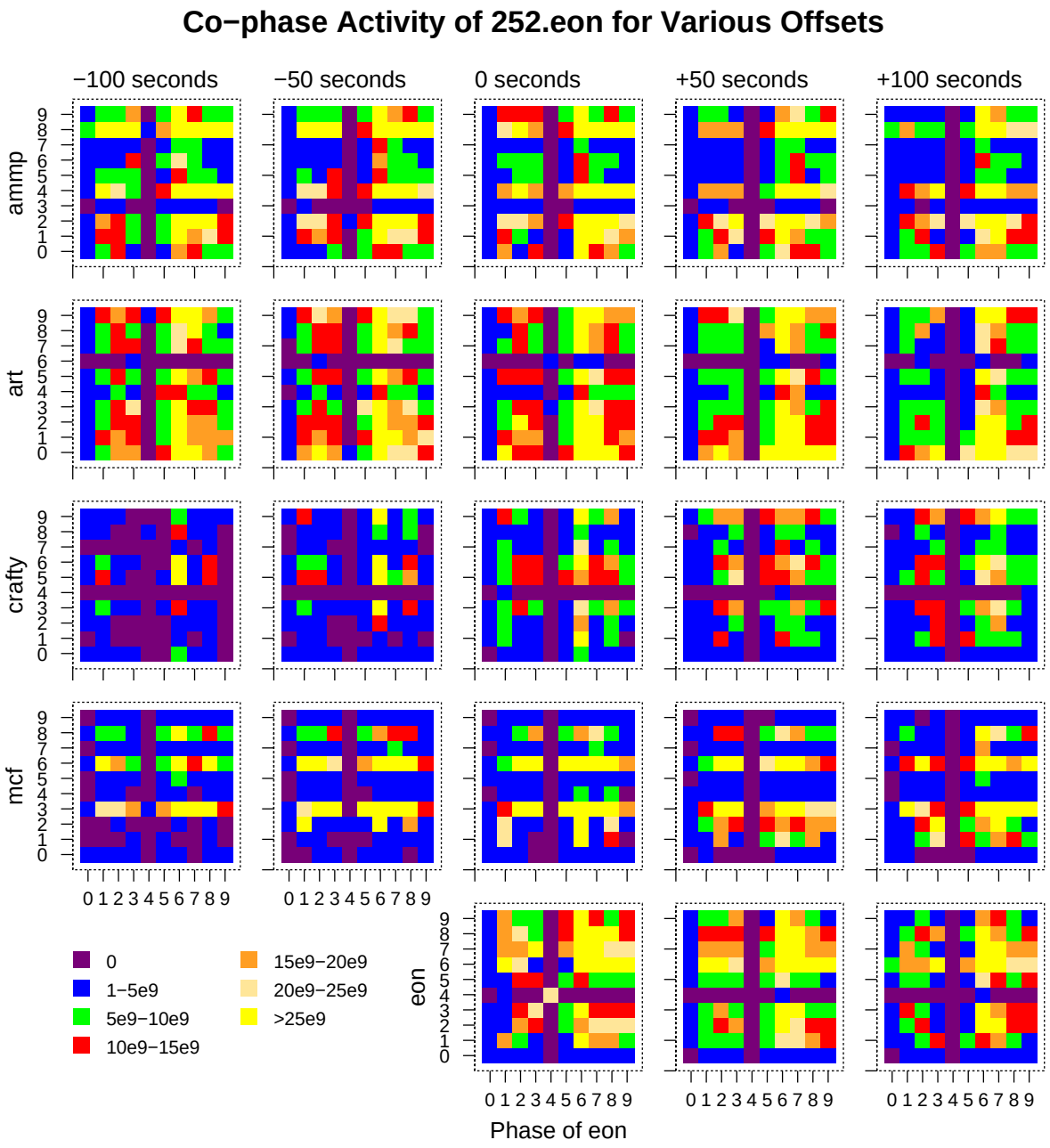

Figure 3. Number of cycles spent in each co-phase of *252.eon* when paired with other *Spec2000* benchmarks and for various offsets. Each row of graphs indicates the benchmark which is paired with *eon* and each column of graphs indicates a different offset. For each graph, the horizontal axis indicates the phase of eon and the vertical axis the phase of the paired benchmark.

threads simultaneously, or with any given single offset, will lead to an incomplete and inaccurate picture. Further, large benchmark suites are such as *SpecCPU* are meant to cover a large number of unique behaviors. By neglecting certain co-phase behaviors, many unique, and possibly very important behaviors may be missed. This paper addresses how to correctly weigh each of co-phases to get the best picture of overall behavior. Further, this gives some initial in-

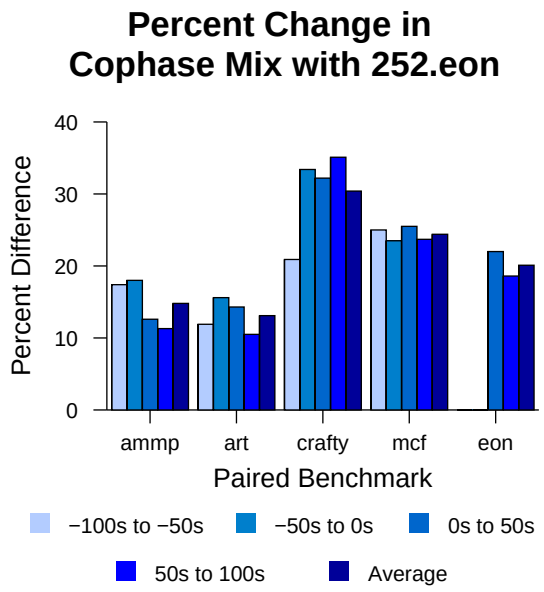

Figure 4. The percent change in co-phase mix due to different start offsets between *252.eon* and four other *Spec2000* benchmarks.

179.art

Co−Phase IPC

Art Phase

Crafty Phase	X	A	B
X	0	.039	.034
A	0	.028	.015
B	0	.034	.013

Interval Series

#	Phase	Length (ops.)
1	A	35.2e6
2	B	43.9e6

186.crafty

Co−Phase IPC

Art Phase

Crafty Phase	X	A	B
X	0	0	0
A	.196	.085	.047
B	.192	.100	.039

Interval Series

#	Phase	Length (ops.)
1	A	142.2e6
2	B	438.2e6

Figure 5. Phase behavior of *179.art* and *186.crafty*. The numbers int the upper tables indicate the IPC of each thread individually for each co-phase combination. The lower tables are the interval series of each thread.

sights into the best manner in which to simulate a multi-threaded system that gives the most accurate picture of execution and minimizes total evaluation time.

3. Illustrative Example

In this section, a simple example program is used to demonstrate the methodology. The example consists of the first two intervals of the *Spec2000* benchmarks *179.art* and *186.crafty*. Execution of each thread is terminated after the second interval in order to keep the example simple. The co-phase and single-threaded information is shown in Figure 5. The upper tables contain the IPC data for each thread in each co-phase, with the table row indicating the phase of *crafty* and the column the phase of *art*. The lower tables in the figure contain the length, in operations, of the intervals of each thread. The data was obtained by using the performance counters on a Pentium-4 processor with Hyper-Threading. The reason for the low IPC numbers is the CISC nature of the x86 ISA.

Since the necessary co-phase performance data is available, it is possible to determine the amount of time that is spent in each co-phase for a given offset. The first step is to determine the performance of the threads before the other thread has started. The offset between the start times of the threads is the amount of time the first thread will spend in single-threaded execution. After the time spent in single-threaded execution is determined, it is possible to determine the time spent in the remainder of the co-phase intervals, in the order [*1.1*], [*1.2*], [*1.x*], [*2.1*], [*2.2*], [*2.x*], [*x.1*], [*x.2*] [1]. The amount of time spent in an interval is determined by the number of operations that need to be performed and the IPC of each thread in that co-phase. When any thread completes all of its operations in its current interval, it will change phase an therefore the co-phase will also change. The number of operations to be completed in a co-phase interval is dependent on how many operations have already been completed in previous intervals, and is therefore a function of start-time offset. The functions for the co-phase intervals of *art* and *crafty* versus offset are shown in Figure 6. In this graph, the x-axis is the offset of the start times of the threads. A positive offset indicates that *179.art* started first and a negative offset indicates *186.crafty* started first. The y-axis indicates the amount of time spent in each co-phase interval. Note that there is no offset such that both intervals [*1.2*] and [*2.1*] are encountered. Since the co-phases are unique to these intervals, no single offset could be used

1 The numbering used here for co-phase intervals is one number per thread, separated by periods. The first number indicates the current execution interval number of the first thread (thread *X*), the second number the current execution interval number of the second thread (thread *Y*), and so on. For the purpose of this discussion, only two threads are considered. Models of more threads are discussed in Section 4.2. An interval number of *0* indicates that a thread has not yet started, and threads marked with an *x* have already completed.

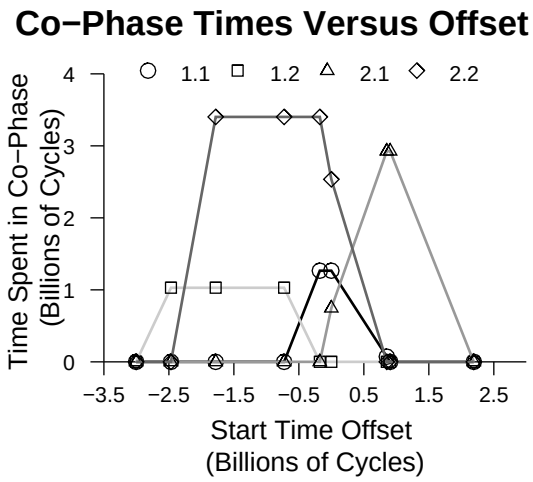

Figure 6. Time spent in the first two co-phases for *179.art* and *186.crafty*.

to characterize both in the same run.

A similar representation of the data can be found in Figure 7. The x-axis is again the offset between the thread start times. However, in this figure the y-axis represents the total execution time. In essence, instead of superimposing the graphs as in Figure 6, the data is now cumulative. There are several interesting aspects of this graph. First, the execution profile, the set of co-phase intervals encountered for a given offset, can be obtained from this graph by simply finding the offset on the x-axis and then following the graph vertically. The graph regions encountered along that vertical line represent the time spent in each interval. The next interesting feature is the top of the graph. The top edge of the shaded regions indicates the total run times versus start time offset. The total run time in this example ranges between $4.78 * 10^9$ cycles and $6.36 * 10^9$ cycles or a range of **34.5%**. The top edge at the left and right extremes of the graph represent the threads being run sequentially, and will always be the same height. For most start time offsets, multithreading is detrimental in that the total run time is higher than running the threads sequentially. This is mainly due to the heavy amount of interference caused by phase **B** of *179.art*. This phase has very poor behavior when paired with either phase of *186.crafty*. The only time that total run time is reduced is when this phase is run predominantly by itself. Although this graph is a good illustration of behavior for a simple example, it is difficult to produce a similar graph for an entire run of a benchmark pairing. Benchmarks can have thousands of intervals that lead to millions of co-phase intervals and therefore graph regions.

Calculating the Average Behavior The final step is to find the areas of each of the co-phase interval region. The area of a region is proportional to the average amount of time that will be spent in that co-phase interval across all possible offsets. The average amount of time spent, or weight, of a region is simply the area divided by the length of sequential single-threaded execution of the threads, which is the range of possible offsets. The computation is fairly simple since each region is simple a series of line segments. The area under the line $mx + b$ between x_1 and x_2 is $\frac{m}{2}(x_2^2 - x_1^2) - b(x_2 - x_1)$. In this example each co-phase has only one interval so there is no summing across intervals to get the total for co-phases. The calculated areas and weights are shown in Table 1. The table also shows the standard deviation in the amount of time spent in each co-phase across the offsets. Although the standard deviation in total run time is only **7.4%**, the deviation in time spent in each co-phase is above 100% in many cases, further demonstrating that any single run will give an incomplete picture.

To characterize average behavior, we simply multiply the weight or area of a co-phase region times the execution characteristics of that co-phase, then divide by the total area of all co-phase regions. The sum of the products across all co-phase regions is the average performance for all offsets. For this example, the average IPC of the multithreaded areas is **0.0713**. Coincidentally, the IPC obtained from a single run with zero offset is a nearly identical **0.0704**, an error of only **1.3%**. However, this is just a fortunate coincidence. In most cases any single run will be quite different from the average. Even in this case, the run time for zero offset is $6.312 * 10^9$ cycles, which is a **7.4%** error from the average of $5.87 * 10^9$ cycles. Additionally, the zero offset run never encounters the co-phase [*1.2*], which is the only co-phase that demonstrates good parallelism in this example.

Co-phase	Area	Average Run Time (Weight)	Standard Deviation
1.1	$1.15 * 10^{18}$	$2.21 * 10^8$	$3.95 * 10^8$(179%)
1.2	$2.35 * 10^{18}$	$4.52 * 10^8$	$4.73 * 10^8$(105%)
2.1	$3.67 * 10^{18}$	$7.06 * 10^8$	$9.59 * 10^8$(136%)
2.2	$8.23 * 10^{18}$	$1.58 * 10^9$	$1.49 * 10^8$(94%)
total	$3.05 * 10^{19}$	$5.87 * 10^9$	$4.76 * 10^8$(8.1%)

Table 1. Co-phase interval region areas and average weights for *179.art* and *186.crafty*

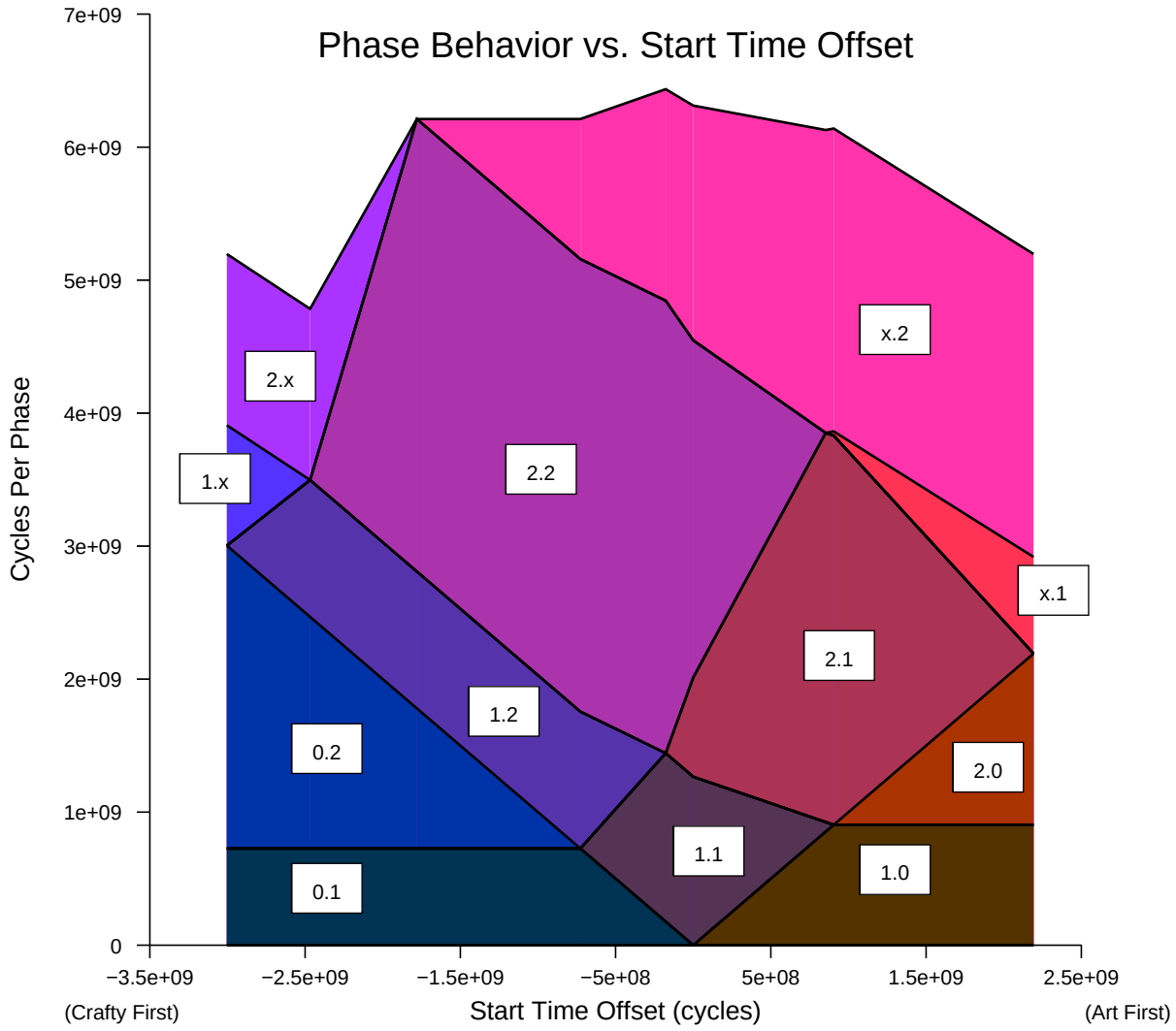

Figure 7. Phase interference behavior of *179.art* and *186.crafty* versus start time offset.

4. Methodology

4.1. Mathematical Model

With the large amount of variation in behavior due to offset, the question is how best to model the overall behavior of the system. Ideally, this also involves minimizing simulation time. The model proposed in this section requires only two inputs. The first is the profile of phase intervals and their lengths for each individual thread and the second is the IPC of each thread in each co-phase, including in each of its single-thread phases. From this data, the expected series of co-phase intervals and their lengths can be derived for any start time offset. More importantly, it is simple to determine the average behavior across all possible offsets as a weighted sum of the behaviors of all of the co-phases.

Once the necessary data is generated, the time spent in each co-phase interval can be determined from the equation in Figure 8. Essentially, the time in an interval will be the shortest possible for one of the threads to complete its current interval. This is only a function of the number of instructions in the interval, the thread's performance in the phase, and the number of instructions that have already been completed from that interval. The number of instructions already completed is the most complicated factor as it is dependent on the performance and the time spent in previous co-phase intervals. For a given interval, this dependency stretches back to all co-phase intervals since the beginning

$$t_{ij} = \min\left(\frac{I_{Xi} - \sum_{k=0}^{j-1} t_{ik} P_{Xik}}{P_{Xij}}, \frac{I_{Yj} - \sum_{k=0}^{i-1} t_{kj} P_{Ykj}}{P_{Yij}}\right)$$

t_{ij} = The number of cycles spent in the *i*th stage of thread *X* and the *j*th stage of thread *Y*.
P_{Xij} = The average number of instructions of thread *X* that are executed per cycle while in the stage combination *ij*.
I_{Xi} = the number of instructions in the *i*th interval of thread *X*.
i,j > 0.

Figure 8. Time spent in co-phase interval i.j.

$$t_{i0} = \begin{cases} 0 & \text{if } t_0 < \sum_{k=1}^{i-1} \frac{I_{k0}}{P_{Xk0}} \\ t_o - \sum_{k=1}^{i-1} \frac{I_{k0}}{P_{Xk0}} & \sum_{1}^{i-1} \frac{I_{k0}}{P_{Xk0}} < t_0 < \sum_{1}^{i} \frac{I_{k0}}{P_{Xk0}} \\ \frac{I_{i0}}{P_{Xi0}} & t_0 > \sum_{k=1}^{i} \frac{I_{k0}}{P_{Xk0}} \end{cases}$$

t_0 = The offset in start times between the threads
t_{i0} = The number of cycles spent in the *i*th stage of thread *X* and in single-threaded mode.
P_{Xi0} = The average number of instructions of thread *X* that are executed per cycle running single threaded in mode in phase *i*.
I_{Xi} = the number of instructions in the *i*th interval of thread *X*.

Figure 9. The time spent in co-phase interval *0i*. Symmetric equations exist for co-phase intervals *0j*

of each thread's current interval. Each of these co-phase intervals in turn is dependent on earlier co-phase intervals, all the way back to the beginning of execution, when only one thread runs because of start time offset. Therefore, the time spent in any co-phase interval is a function of start time offset. In the equation in Figure 8, the portion of the function where the thread X term controls the *min* function, the next co-phase interval will be the next interval of X. Obviously, the same can be said for thread Y. The two terms of the function will only intersect once because although the slope of the line changes many times, the term for one thread will be monotonically increasing versus offset and the other will be monotonically decreasing. This makes sense as the greater the offset is, the further the thread will be in its execution. The intersection of the two point marks the boundary between which co-phase interval will occur next or the point where both phases will end simultaneously.

The minimum function makes it very difficult to create closed form equations for any intervals past the first few intervals. Such functions exist, since any finite execution will have a finite number of co-phase intervals, but they can become prohibitively large very quickly. The number of possible predecessor intervals is proportional to the number of intervals each thread has completed. Since each of these have their own predecessors, the total number of terms in a given equation becomes very large very quickly. Instead, it is much easier to solve the equations numerically. That is, for a given system, find the function of the offset time that determines the amount of time each interval will occur by adding the functions of its predecessors, starting from the intervals that have no predecessors. The only co-phase intervals that have no predecessors in a two thread system are interval [*0.1*] and [*1.0*]. Co-phase interval [*0.1*] can end in one of three ways: either thread X will start and the co-phase interval will become [*1.1*], the interval of thread Y will complete and the interval becomes [*0.2*], or thread X will start exactly as thread Y finishes its interval and the co-phase interval becomes [*1.2*]. The amount of time spent in interval [*0.1*] is equal to the start time offset up to the maximum of the time it takes interval 1 of thread Y to complete running alone. The equation for the amount of time spent in co-phase interval [*i.0*] is shown in Figure 9. With this data from interval [*0.1*], the function for [*0.2*] can be derived. This is repeated for all intervals of thread Y. The process is then repeated for thread X. Once the functions for co-phase intervals [*0.1*] and [*1.0*] are determined, it is possible to derive the function for co-phase interval [*1.1*] using the equation in Figure 8. All of the execution time function of each co-phase interval functions can be derived numerically once all of the functions of all of their predecessors are derived. The simplest order to derive the function is using a nested *for* loop. That is, derive all of the functions for co-phases [*0.j*] for j starting at 1 to the last phase of thread Y, then for [*1.j*] for j starting at 0 and then for each interval of thread X. After all of the interval functions have been derived, the average behavior can be calculated as explained in section 3.

A few key assumptions are made for this model. The first assumption is that co-phase behavior follows the phase behavior of individual threads. This means that the length of the execution intervals will remain constant, in terms of total operations executed between single and multithreaded modes. The reason that this may change is that a phase is the set of periods with similar but not identical behavior. Interference in the multithreaded environment may affect different periods differently, effectively splitting up a phase or making periods that are in different phases in single threaded execution behave in similar ways when multithreaded. This is an area of ongoing research. A second related assumption is that co-phase behavior is consistent in that the behavior in a given co-phase is the same between intervals and within each interval. This assumption must be made anytime phase analysis is used. The final assumption is that no accounting is made for threads switch-

ing out during execution in the model. The assumption is not that this switching will not occur, but that the net effect of such switches will be zero. A change in thread being switched out and then switched back in can be thought of as the simulation stopping when the thread is switched out, then restarting at the same execution time with a new offset. In effect, the thread that is not switched out is fast forwarded relative to the thread which was switched out. The assumption is that either thread is equally likely to be switched out for an equal amount of time. Over the totality of all tests, the effects of switch outs cancel each other out. The overhead of the context switch and of the thread warming up when it is switched back in are neglected because this will be small compared to the overall execution time. The direct and indirect costs of context switches are examined in [7].

4.2. Extension to More Than Two Threads

The situation becomes considerably more complex when more threads are used. The first complication is that the number of co-phases and co-phase intervals grows exponentially with the number of threads. Dealing with this problem is a matter of predicting which co-phases will contribute the most to the final results and have interesting interference. This is the subject of ongoing research.

Next, representing the data becomes more difficult. The number of possible offset variables between any two threads is the factorial of the number of threads. This problem is solved by defining the start time of one thread as a zero point and defining all other start times relative to this one. The relative start times of any two threads can be easily determined from their offset from the reference.

It would seem that the choice of which thread is chosen as the reference would affect the projection and thus area and weights between the regions. Fortunately the areas are constant no matter which representation is chosen. The matrix representation of the transform between representations is shown in Figure 10. That is, in order to move a point from the representation where all offsets are defined in terms of thread **1** to one where the offsets are all defined in terms of thread **2**, the point is represented in column vector form and multiplied by this matrix. It can be shown that $R_n^{n+1} = I_n$. In other words, applying the transform in an *n*-thread system *n+1* times yields the original representation. A formal proof is beyond the scope of this paper, but the basic idea is that since any change in area produced by the transform would cause a change in the representation once it was repeatedly applied, it follows that each application of the single transform does not change the areas. More information on transforms can be found in [9].

Fortunately, the equation in Figure 8 is still valid if terms are added for the additional threads. However, the equation in Figure 9 becomes somewhat more complicated with more threads but the basic idea is the same. The region which represents a given startup co-phase interval will start at the point where that co-phase is reached, based on previous startup intervals. The region ends where another thread starts or the interval of that thread ends.

$$R_n = \begin{bmatrix} -1_{(n-1)x1} & I_{n-1} \\ -1 & 0_{1x(n-1)} \end{bmatrix}$$

Figure 10. The matrix for transforming representations between reference threads (I_n is the n-dimensional identity matrix).

5. Conclusion

5.1. Future Work

The accuracy of multithreaded simulation is vitally important to the development of systems which are becoming increasingly ubiquitous across the industry. However, also important is the speed of these simulations. Because of inter-thread interactions, the number of unique behaviors of a multithreaded system is the product of the number unique behaviors in each thread. This further exacerbates the problem of long simulation times. Although this paper has shown that accurate simulation of multithreaded systems requires that the entire co-phase space be covered, it is impractical in all but the most trivial cases to simulate all co-phases for their duration. The next step in our research involves developing methods to efficiently cover that space. This entails applying modern sampling techniques such as SMARTS [18] and SimPoint [11] to multithreaded environments to reduce simulation times. Along these lines, we are also investigating techniques to increase parallelism in simulation to improve overall simulation latency. Additionally, we are developing methods to reduce the effective size of the co-phase space without sacrificing accuracy. We are investigating techniques to predict multithreaded behavior from single threaded behavior and determining which co-phases will have the most important and interesting behavior to reduce the number of co-phases which must be simulated.

5.2. Summary

Multithreaded architectures, in various forms, are an increasingly popular technique for circumventing the bottlenecks of modern processors. The interference between

threads, however, has a dramatic impact on overall performance. Interference is dictated by how the individual threads interact in their various phases. Since phase interaction is determined by the relative position of the threads, this further complicates the characterization of multithreaded systems. The experimental results presented in this paper demonstrate that this offset has a substantial effect on overall performance. The model presented balances the importance of individual co-phases and allows more accurate modeling of real world performance of multithreaded architecture, where effects such as OS scheduling cause random offsets between threads. This increase in the correlation between simulated and achieved performance is vital to the effective and efficient design of future multithreaded architectures.

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfi eld, and B. Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, 1990.

[2] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 266–277. ACM Press, 2001.

[3] A. S. Dhodapkar and J. E.Smith. Comparing program phase detection techniques. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 217–227, 2003.

[4] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–26, / 1997.

[5] Intel Corporation. *Intel Pentium 4 Processor with 512-KB L2 Cache on 0.13 Micron Process and Intel Pentium 4 Processor Extreme Edition Supporting Hyper-Threading Technology Datasheet*. Santa Clara, CA, 2004.

[6] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *Proceedings of the Third IEEE Annual Workshop on Workload Characterization*, pages 102–110, 2000.

[7] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 75–84. ACM Press, 1991.

[8] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *Proceedings of the 37th Annual Symposium on Microarchitecture (Micro-37 2004)*, 2004.

[9] L. Sadun. *Applied Linear Algebra: The Decoupling Principle*. Pearson Education, Prentice Hall, Upper Saddle River, NJ, 2001.

[10] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to fi nd periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2001.

[11] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *In Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[12] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 17th annual international conference on Supercomputing*. ACM Press, 2003.

[13] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[14] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 414–425, 1995.

[15] R. Steven E and S. K. Reinhardt. The impact of resource partitioning on smt processors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2003.

[16] M. VanBeisbrouk, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, 2004.

[17] W. Weber and A. Gupta. Exporing the benefi ts of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proceedings of the 16th Interational Symposium on Computer Architecture (ISCA)*, pages 273–280, June 1989.

[18] R. E. Wunderlich, T. F. Wenisch, B. Falsafi , and J. C. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30nd Annual International Symposium on Computer Architecture (ISCA)*, pages 84–95, 2003.

[19] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *The 11th International Symposium on High-Performance Computer Architecture (HPCA)*, February 2005.