Scheduling Task Dependence Graphs with Variable Task Execution Times onto Heterogeneous Multiprocessors

Nadathur Rajagopalan Satish
Kaushik Ravindran
Kurt Keutzer

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-42
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-42.html

April 21, 2008

Scheduling Task Dependence Graphs with Variable Task Execution Times onto Heterogeneous Multiprocessors

Nadathur Satish[*]
Electrical Engineering and Computer Sciences
UC Berkeley

Kaushik Ravindran[†]
Electrical Engineering and Computer Sciences
UC Berkeley

Kurt Keutzer
Electrical Engineering and Computer Sciences
UC Berkeley

ABSTRACT

We present a statistical optimization approach for a scheduling a task dependence graph with variable task execution times onto a heterogeneous multiprocessor system. Scheduling methods in the presence of variations typically rely on worst-case timing estimates for hard real-time applications, or average-case analysis for other applications. However, a large class of soft real-time applications require only statistical guarantees on latency and throughput. We present a general statistical model that captures the probability distributions of task execution times as well as the correlations of execution times of different tasks. We use a Monte Carlo based technique to perform makespan analysis of different schedules based on this model. This approach can be used to analyze the variability present in a variety of soft real-time applications, including a H.264 video processing application.

We present two scheduling algorithms based on statistical makespan analysis. The first is a heuristic based on a critical path analysis of the task dependence graph. The other is a simulated annealing algorithm using incremental timing analysis. Both algorithms take as input the required statistical guarantee, and can thus be easily re-used for different required guarantees. We show that optimization methods based on statistical analysis show a 25-30% improvement in makespan over methods based on static worst-case analysis.

1. INTRODUCTION

The increasing complexity of designing conventional processors and increasing power consumption has resulted in a shift towards chip multiprocessors. The key to successful application deployment on these multiprocessors lies in effectively mapping the concurrency in the application to the architectural resources provided by the platform. Static compile time task allocation and scheduling is an important step during the mapping process. Static models and methods become viable when the application workload and parallel tasks are known at compile time. Static scheduling has been used in the mapping of media and signal processing applications [17]. It is also useful in rapid design space exploration for micro-architectures and systems [11].

In this work, we consider the problem of statically scheduling a graph of concurrent tasks with associated execution times on a target heterogeneous multiprocessor. The objective of the scheduling algorithm is to minimize the schedule length (makespan) of the system. Static models and methods rely on exact knowledge of the execution time of different tasks at compile time. However, real task execution times can vary significantly across different runs due to (1) loops and conditionals in the code leading to different execution traces and (2) variation in memory access times due to cache accesses and bus arbitration [20]. Static scheduling algorithms in the presence of variations rely on worst-case behavior for hard-real time applications [2], and average-case behavior for non real-time applications. In contrast, a large class of recent applications fall under the category of soft real-time applications. Examples include applications in the domains of multimedia (video encoders/decoders), networking and gaming applications. Such applications do not require hard guarantees on latency or throughput, but do require statistical guarantees on steady-state behavior. For example, a H.264 video application may require that 95% of all frames are decoded within a specified latency. For such applications, a static scheduling method may does not best utilize system resources. Dynamic scheduling is an alternative, but the run-time overhead can be prohibitive [17]. This motivates a move to statistical models and scheduling methods for effectively mapping soft real-time applications.

In this paper, we present a statistical model and optimization methods for scheduling concurrent applications onto heterogeneous multiprocessors. Our model involves using general joint probability distribution tables to capture variations in task execution times. Our work was motivated by the variability found in the H.264 video decoding application, but the model is sufficiently broad to capture a variety of soft real-time applications. We use a standard Monte Carlo technique to analyze and compare schedules derived from this statistical model. Monte Carlo analysis is considered a "golden" model for statistical analysis and can handle any type of variations, but is usually compute intensive. Our use of Monte Carlo analysis is made possible due to the following factors: (1) most scheduling problems only involve hundreds of random variables as opposed to other systems in financial analysis and circuit timing analysis involving thousands to millions of variables, (2) we can perform an incremental Monte Carlo analysis to speed up the computation.

We are interested in computing the best schedule for a concurrent application. Our target platform is a heterogeneous multi-core architecture. This optimization problem is known to be strongly NP-hard even for static models and homogeneous architectures [9]. A variety of methods have been proposed for scheduling static models onto multiprocessors. Kwok and Ahmed present a comprehensive taxonomy of heuristic methods [16]. Heuristic approaches work well for multiprocessor scheduling problems that do not consider a variety of practical constraints. However, scheduling algorithms that handle constraints such as specific architecture topologies, memory limitations and so on have to deal with a complicated and not very well understood solution space. A

[*]Contact author
[†]Now at National Instruments Inc.

general method that has been successfully applied to practical and complex multiprocessor scheduling problems is Simulated Annealing[14][21].

In this work, we propose two algorithms, one a list scheduling based heuristic that uses Monte Carlo analysis and the other a simulated annealing technique that utilizes incremental statistical analysis to schedule a concurrent application with statistical execution times to a multiprocessor. Both our algorithms take in the required statistical guarantee as an input. Thus the same algorithms can be re-used for different required statistical guarantees.

In order to demonstrate the effectiveness of our method, we compare the results of statistical scheduling for different guarantees to static scheduling with worst-case execution times. We are typically only interested in high guarantees, and therefore do not consider scheduling with average-case execution times. We show that the static algorithm with worst-case execution times makes wrong scheduling decisions due to a wrong estimation of task completion times. On the other hand, the statistical methods use the Monte Carlo analysis for a given schedule and hence makes more accurate decisions.

2. RELATED WORK

The problem of mapping task-level concurrency to a multiprocessor architecture has been studied since the 1960s. Methods used to solve this problem usually assume that the task execution times are fixed. A classification of different heuristic methods to solve the scheduling problem can be found in [16]. One popular heuristic approach is list scheduling for which different variants have been proposed [12, 3, 13, 23]. Methods based on branch-and-bound, evolutionary algorithms and constraint formulations have also been devised [25]. Simulated annealing has recently found success in multiprocessor scheduling problems [21]. Ravindran [22] shows a toolbox of methods for solving the static scheduling problem.

In the case when task execution times vary, different models for accounting for the variations have been proposed based on both analytical and simulation-based techniques [26, 10]. In this work, we adopt a general simulation-based model that can capture any general variability in code. The impact of such execution time variability on scheduling real-time periodic tasks with deadlines onto multiprocessors has been studied in [19]. This method assumes that tasks are periodic and have individual deadlines. In this work, we focus on scheduling aperiodic tasks with stochastic execution times onto multiprocessors to optimize for schedule length (makespan). Since the makespan is a distribution, we are interested in a particular percentile of the distribution.

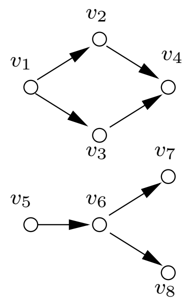
Statistical timing analysis has been well-studied recently for circuit timing in the presence of process variations [8]. Circuit timing analysis involves thousands to millions of gates and a fast analysis is paramount. The models used typically assume normal or near-normal delay distributions and the analysis is usually analytical. In contrast, task execution times are often not normally or even continuously distributed and are not easily amenable to analytical analysis. We use a general Monte Carlo analysis in view of the limited problem sizes of at most a few hundred tasks.

Statistical optimization has been primarily used for power optimization [24] and gate sizing [18]. These problems are usually modeled as convex optimization problems. Scheduling, on the other hand, is an inherently discrete and non-convex problem. Compilers have used profiling to drive instruction scheduling [4], but these typically based on average-case analysis. We target applications where high statistical guarantees are required. Statistical optimization has, however, been used for scheduling job-shop applications in the Operations Research community [1]. The work in [1] solves a special case of the multiprocessor scheduling problem we consider, and only uses the means and standard deviations of jobs. In this work, we consider arbitrary distributions of task execution times.

3. PREVIOUS WORK: STATIC MODELS AND METHODS

In this section, we consider a representative static scheduling problem. We introduce terminology and concepts that will be used in our statistical scheduling work as well.

3.1 Static Scheduling Problem



	$w(v)$
v_1	3
v_2	3
v_3	1
v_4	7
v_5	1
v_6	2
v_7	5
v_8	6

Figure 1: An example task dependence graph with annotations for execution times of each task.

We consider a static scheduling problem whose objective is to schedule a task dependence graph onto a multiprocessor to minimize the schedule length (*makespan*). The task graph is a directed acyclic graph (DAG) $G = (V, E)$, where $V = \{v_1, v_2, ...v_n\}$ represents the set of computation tasks, and the directed edges $E \subseteq V \times V$ represent dependence constraints between the tasks. The architecture graph $H = (P, C)$ is a graph showing the processors and their interconnections, where $P = \{p_1, p_2, ...p_m\}$ is a set of processors and $C \subseteq P \times P$ shows the connectivity between the processors. Each task is fully executed without preemption on a single processor. The performance model is represented by $W : (V \times P) \cup (E \times C) \rightarrow \Re^+$, where $W(v, p)$ is the execution time of task v on processor p and $W(e, c)$ is the communication latency of edge e on communication link c. An example task dependence graph with the performance model is displayed in Fig. 1. For convenience, we usually add two dummy nodes with zero execution time, a *source* node with edges to all nodes without a predecessor and a *sink* node with edges from all nodes without a successor.

For a task dependence graph $G = (V, E)$ and an architectural model $H = (P, C)$, we define a valid *allocation* as a function $A : V \rightarrow P$ that assigns every task in V to a single processor in P. Given a particular A, the communication delay between tasks v_1 and v_2 is given by $W((v_1, v_2), (A(v_1), A(v_2)))$, where (v_1, v_2) represents the edge between tasks v_1 and v_2, and $(A(v_1), A(v_2))$ represents the communication link between processors $A(v_1)$ and $A(v_2)$. A valid *schedule* is defined as a function $S : V \rightarrow \Re^+$ that assigns a non-negative start time to each task, and satisfies two sets of constraints.

$$\begin{aligned}
&\forall(v_1, v_2) \in E(\text{dependence constraints}),\\
&(a)\ S(v_2) \geq S(v_1) + W(v_1) + W((v_1, v_2), (A(v_1), A(v_2)))\\
&\forall v_1, v_2 \in V, v_1 \neq v_2(\text{ordering constraints}),\\
&(b)\ A(v_1) = A(v_2)\ \Rightarrow\\
&\qquad S(v_1) \geq S(v_2) + W(v_2) \vee S(v_2) \geq S(v_1) + W(v_1)
\end{aligned}$$

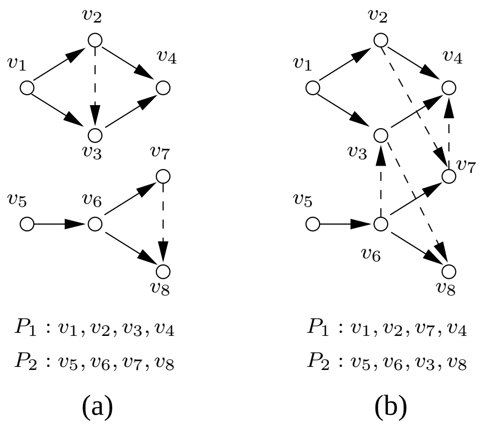

v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8
$P_1 : v_1, v_2, v_3, v_4$
$P_2 : v_5, v_6, v_7, v_8$
(a)
$P_1 : v_1, v_2, v_7, v_4$
$P_2 : v_5, v_6, v_3, v_8$
(b)

Figure 2: Two valid allocations and schedules for the task dependence graph in Fig.1 onto two processors. The dotted lines represent ordering edges and the solid lines are dependence edges. The longest path gives the *makespan* of the schedule.

Constraint (a) enforces that the task dependencies are met: a task starts only after all predecessors are complete. Constraint (b) enforces a total ordering among the set of all tasks assigned to a single processor.

The *makespan* of a valid allocation and schedule is defined as $\max_{v\in V} S(v) + W(v)$ or alternatively, as $S(snk)$ where snk is the sink node. The problem of computing the makespan is called the *analysis* problem. Given a graph $G(V, E)$, an allocation A and a schedule S, we can capture the schedule graphically by adding additional edges to the task graph. Define $E'' = \{(v_1, v_2) \notin E | A(v_1) = A(v_2) \wedge S(v_2) \geq S(v_1) + W(v_1)\}$ to be the set of *ordering* edges that define the order in which tasks allocated to the same processor execute. Define $G' = (V, E')$ where $E' = E \cup E''$. G' is the graph G with the ordering edges added. G' is a DAG for a valid schedule. Analysis of a schedule is then a longest path computation on G', which is a breadth-first traversal of the graph G'. Fig. 2 gives two valid allocations and schedules for the task graph in Fig. 1 scheduled on two processors. The dotted lines in Fig. 2 depict the ordering edges. The objective of the scheduling problem is to compute a valid allocation and schedule with *minimum makespan*.

4. STATISTICAL MODELS AND ANALYSIS

In many real world examples, static models are insufficient to accurately reflect execution characteristics of tasks. There are usually variations in task execution times due to the presence of conditionals or loops inside code, and memory access time variations due to cache misses and bus arbitration The task execution time is then properly expressed as a distribution rather than a single value. We introduce a statistical model for task execution times, and show techniques to perform analysis and optimization on this model.

4.1 Timing Model

There have been previous attempts at characterizing the execution time of tasks. These can be classified into analytical and simulation based approaches. Analytical approaches build up the variability of a task from the variability of each statement of code and the structure of the code. These can be accurate but are usually difficult to build. Simulation based approaches execute the code with different inputs and get the distribution of real execution times. Such an approach is simpler but assumes access to the final platform.

In this work, we use a simulation-based approach to characterize the run times of individual tasks. There are two main types of variations we wish to capture: (1) variations in the runtime of a single task across many runs due to cache effects or bus arbitration (2) variations in the runtime due to different inputs exciting different execution traces within the task. To capture the second effect, we simulate the runtimes of each task in the task graph with different inputs. We repeat the execution of each input a certain number of times to ensure that variations in cache access times are captured in our traces. The execution times of each task in each of these runs are then discretized by binning and stored in the form of a probability distribution table. Such a table has entries of the form (runtime range, probability), and stores the probability that the task has an execution time in a particular range. If we know that the task execution times are independent, then this is all we need. If the task execution times are instead dependent, which is the more general case that arises mainly due to global sources of variation having to do with varying memory access times, then we must store the joint probability distributions. The joint probability distribution is also stored in the form of a table, but the entries are of the form ($range_1$, $range_2$, ... , $range_n$, probability) that stores the joint probability that the task execution times of task 1 lies in range 1 and the execution time of task 2 lies in range 2 and so on. It may so happen that only certain sets of tasks have dependent task executions, while others do not. In this case, the joint probability distribution tables only need contain the dependent variables. In general, the joint probabilities help capture the correlations between the execution times of different tasks. This method of capturing task execution time variations is very general and can capture any type of variations.

4.2 Example: H.264 decoding

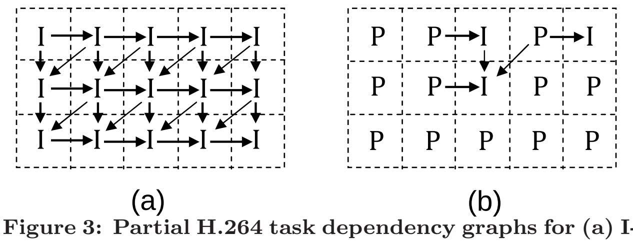


Figure 3: Partial H.264 task dependency graphs for (a) I-frames having only I-macroblocks and (b) P-frames with both I- and P-macroblocks.

The need for such a general model is made evident in the context of decoding of H.264 macroblocks. In previous work [5], we have discussed the parallelization of H.264 decoding on multiprocessor architectures. The parallelization is across different macroblocks within a single frame. Task graphs for parts of H.264 frame decoding can be seen in Figure 3. Each task is responsible for the decoding of a single macroblock. There are two main sources of variations in execution time.

1. Variations in macroblock type: We consider only I and P-frames below. An I-frame (or intra frame) consists of tasks decoding I(intra)-macroblocks. These macroblocks usually depend only on other macroblocks within the same frame, and some residual information. A P(predicted)-frame contains both I and P-macroblocks. Depending on the input, a particular task may be responsible for decoding I or P macroblocks. P-macroblocks depend on previously decoded frames as also residual information. I and P macroblocks have different execution time characteristics (Fig 4). There is variability in the execution time of both I- and P-macroblocks due to varying amount of residual information. For further details, we refer the user to [5].

2. Global frame buffer access: An important source of variations in H.264 arises because each P-macroblock has to access a global shared frame buffer. Cache misses and/or bus arbitration leads to uneven execution times even when a single P-macroblock is executed repeatedly with the same input. Since each P-macroblock has to access the frame buffer, this is a global source of variation in task execution times, and causes task execution times to be correlated.

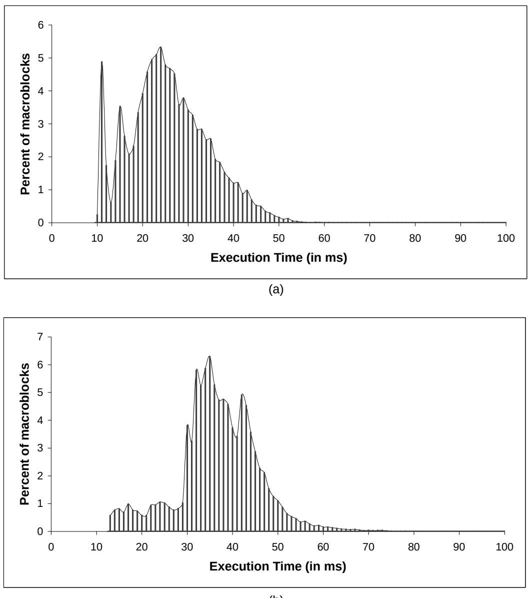


Figure 4: Histogram of the execution time variations with (a) I-macroblocks and (b) P-macroblocks of the manitu.264 video stream in H.264 decoding.

The overall run time of each individual task is a complex function of the input macroblock type and cache access times. There is a further complication in that the total cache access time depends on the number of accesses to the global frame buffer which is, in turn, dependent on the type of macroblock (only P-macroblocks depend on past frames and need to look at the frame buffer). Thus to summarize: there is execution time variation between I- and P-macroblocks, there are execution time variations due to the different amount of residual information in different I- and P-macroblocks, and there is a global source of variation that is only applicable to P-macroblocks due to global memory/cache access times. Modeling this system analytically will very likely prove difficult. Our simulation based scheme can capture all these sources of variation. We have individual probability distribution tables for each I-macroblock that take into account the residual information. We have a joint probability distribution for all the P-macroblocks that take into account both the individual variations due to residual information as well as the global frame buffer access times. Figure 4 shows the nature of the probability distributions of execution times for the I-macroblocks and P-macroblocks for the manitu.264 video stream. For the sake of depiction, the figure does not show the joint probability distribution for all P-macroblocks in the frame. It instead shows the marginal probability distribution for a single P-macroblock (eliminating all other variables from the joint distribution through summation).

4.3 Statistical Analysis

For the statistical timing model, a valid schedule computes a start time probability distribution for each task in V, subject to the constraints (a) and (b) in Section 3.1.

The *makespan* of a schedule is defined as $\max_{v \in V} S(v) + W(v)$. The makespan is no longer a single number as in the static case, but is instead a distribution. Given a distribution and a number η ($0 \leq \eta \leq 100$), the η'th percentile of the distribution is defined as the value below which η% of the observations fall. For a given schedule, the η'th percentile of the makespan is the value that gives us a statistical guarantee that no more than $100 - \eta$ % makespans out of repeated simulation runs of the schedule will exceed it. This may be likened to the concept of yield in timing analysis of circuits. The objective of statistical optimization is to compute the schedule with the minimum η'th percentile of makespan distribution. For an H.264 application, this corresponds to optimizing for makespan while guaranteeing a certain quality of service, viz. that η% of the frames will finish before the makespan.

To perform the analysis, we use the notation of "ordering" edges introduced in Section 3.1. The problem of computing makespan reduces, as in the static case, to a longest path computation on the graph with ordering edges. However, we now have to compute the longest path in a graph where each node execution time is an arbitrary random distribution expressed by a (individual or joint) probability distribution table. In this general case, analytical approaches to compute the longest path in the presence of correlations need to consider all permutations of entries in each table, which is infeasible. In this work, we use a Monte Carlo simulation approach to compute the longest path. The Monte Carlo analysis is simple in concept: we take a sample for each independent variable (either an entry in the joint probability distribution table or one for each individual probability table). Using these samples as deterministic values for the random variables, we perform a deterministic longest path analysis on the graph. We then repeat this experiment a number of times to get the final distribution. Once we compute the probability distribution of the makespan, it is simple to obtain the η^{th} percentile. The only complication comes when there are multiple joint probability distribution tables for subsets of tasks, and there are shared tasks across tables. In this case, we need to pick consistent values for variables across different tables.

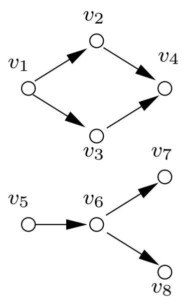
A Monte Carlo simulation usually requires a number of iterations to converge to a distribution, incurring significant computation and execution time. The number of iterations required for obtaining a good estimate depends on the number of variables and the nature of the variations. In our work, we typically deal with small task graphs with only a few hundred variables. Thus, we have found that doing a set of 1000 Monte Carlo iterations is sufficient. For efficiency, we generate all 1000 samples for each task and then do a single breadth-first graph traversal for computing all longest paths. We can further improve computation time through our incremental analysis as outlined in Section 5.

4.4 Deterministic analysis

We are typically interested in solving the analysis problem at a high value of η to provide high statistical guarantees on makespan. One deterministic approach to analyzing a statistical task graph is to use a worst-case estimate on the execution time of each individual task, and then perform a normal static longest path analysis (Sec. 3.1). The worst-case estimate of a task is obtained by taking the 99.9'th percentile (or some other suitably chosen percentile) of its execution time distribution. This method is conservative since there is very little chance that all individual random variables simultaneously attain their extreme values.

Another deterministic approach is to use the average-case execution times of individual tasks. This approach is not conservative, but can heavily underestimate the final makespan if we are interested in high percentiles. Soft real-time applications typically require high guarantees on makespan, and thus average-case analysis is unsuited.

In the next section, we show that the statistical analysis can be much more accurate than deterministic worst-case analysis.



	Mean	*St.Dev.*
v_1	3	0.2
v_2	3	0.3
v_3	1	0.3
v_4	7	0.1
v_5	1	0.6
v_6	2	0.7
v_7	5	0.7
v_8	6	0.7

Figure 5: The task dependence graph in Fig. 1 with annotations for the mean and standard deviation of execution times of each task.

4.5 Validation of Analysis

In this section, we show with the help of a simple example that deterministic analysis can be very inaccurate as compared to the Monte-Carlo statistical analysis.

Consider the simple task graph in Fig. 5. For this example, we assume that the execution times of tasks are independent and are normally distributed. The means and standard deviations of task execution times are shown in the figure. In our model, we would discretize these values and have a table representing the probability distribution of each task. We do not show this table here for the sake of clarity. Assume that the task graph is to be scheduled on a homogeneous two-processor system.

For the two schedules shown in Fig. 2, we run Monte Carlo simulations of the scheduled graph to obtain our golden model of makespan distributions. For $\eta = 99\%$, we compare the makespan percentile computed by the statistical analysis with the deterministic estimate obtained by individually raising the execution time of each task to its worst-case estimate. This is shown in Fig. 6 for the two schedules. The same figure also shows the deterministic estimate obtained by taking the average execution time for each task.

Fig. 6 shows that the deterministic worst-case analysis overestimates the Monte Carlo makespan at η=99% in both schedules (a) and (b). Further, the worst-case analysis does not overestimate the two schedules equally: it is more accurate for schedule (b) than for schedule (a). In fact, the worst-case analysis would conclude that schedule (b) is better than schedule (a), whereas the Monte Carlo simulations indicates that schedule (a) is better than schedule (b). As a general rule, worst-case estimates are inaccurate if critical and near-critical paths have high variance, which is the case in schedule (a). In schedule (b), these paths are of low variance, and hence the worst-case estimate is more accurate. Accuracy can vary significantly across schedules. Fig. 6 also shows that the deterministic average-case analysis very significantly underestimates the Monte Carlo makespan. Using average-case analysis is tantamount to scheduling for close to the average-case scenario, providing a very low guarantee on makespan. In cases where we are interested in very high guarantees on makespan, it is not usually advisable to utilize average-case analysis. We do not further consider average-case analysis in this work.

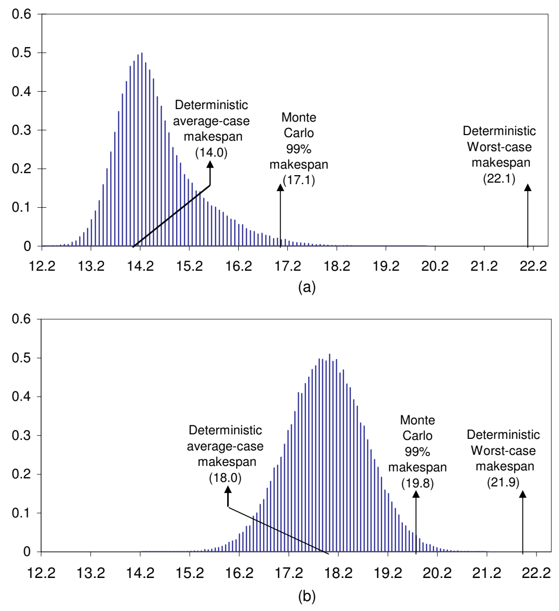


Figure 6: Comparison of Monte Carlo analysis versus deterministic analysis for schedules in Fig. 2(a) and (b) resp. The worst-case and average-case deterministic makespans and the statistical makespan for η=99% are shown. Both worst-case and average-case analysis are inaccurate.

The above trends are consistent for larger examples as well. In particular, the worst-case deterministic analysis consistently overestimates the makespan, while the percentage difference between the worst-case analysis and Monte Carlo analysis varies considerably across different schedules. A histogram of the percentage differences between the deterministic worst-case estimate and the 99% percentile of a Monte Carlo makespan distribution for a set of 1000 ran-

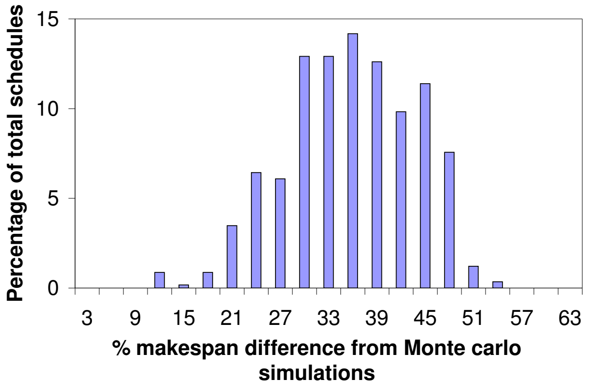


Figure 7: Histogram of the percentage differences between worst-case deterministic analysis and Monte Carlo analysis for a set of 1000 schedules for H.264 decoding on manitu.264. The deterministic analysis is uneven in its overestimation of makespan.

domly chosen schedules of a task graph for decoding a H.264 video stream is shown in Fig. 7. There is a large spread in the accuracy of the deterministic worst-case estimates with respect to the Monte Carlo values. Any optimization method based on this analysis is thus significantly suboptimal.

Monte Carlo analysis is a "golden" model and is considered very accurate. The major problem with it is that it is slow and unsuited to be in the inner loop of an optimization method. In order to speed up our Monte Carlo analysis, we developed an incremental statistical analysis that we describe next.

5. INCREMENTAL STATISTICAL ANALYSIS

The Monte Carlo statistical analysis method will be used inside the inner loop of a Simulated Annealing Engine in one of our optimization methods. Performing a full Monte Carlo analysis for each iteration of the Simulated Annealing loop can become computationally infeasible.

An incremental analysis is useful when we already have an analysis for a graph, and the graph structure or node timing values is then perturbed a little. Incremental analysis has been studied in the context of circuit simulation and physical synthesis. In the context of statistical scheduling, we shall see in Section 6.2 (Fig 8) that only a few "ordering" edges change across simulated annealing iterations, while all timing values for individual nodes remain the same.

The addition and removal of edges in a graph can change the arrival times of only the nodes in the fanout cone of the modified edges. All random values used in for the other nodes are unchanged from the existing analysis. This kind of limitation in the nodes to be considered for analysis has been called level limiting analysis in [8]. In our statistical analysis, we perform (for efficiency reasons) longest-path analysis for all iterations using a single breadth-first search of the graph. In the incremental analysis, we do the same – except that we only need to perform a breadth-first traversal of the graph starting from the target nodes of the modified edges.

Another facet of incremental analysis is dominance-limiting analysis. If a particular modified edge does not affect the arrival time of its target node, then it has been dominated by the node's other edges. Then we need not proceed with a breadth-first traversal of that edge. In the context of our Monte Carlo analysis, we perform a number of deterministic runs. If most of these runs (a fraction above a certain threshold) are not affected by the modified edge, then we do not need to propagate arrival times through that node.

These two techniques together significantly reduce the number of nodes to be considered for analysis. Since the number of random variables decreases, the Monte Carlo analysis also tends to converge with fewer iterations.In the Simulated Annealing algorithm, we obtain a speedup of 2-3X over performing full Monte Carlo analysis.

6. STATISTICAL OPTIMIZATION

For a given allocation and schedule, Section 4.3 shows us how to compute the makespan distribution and the η'th percentile of makespan. The problem of finding the valid allocation and schedule that achieves a makespan distribution with the *minimum η'th percentile* for a given η is the *optimization* problem.

The problem is known to be strongly NP-hard, even in the special case of deterministic execution times. As such, various heuristic [16] and exact algorithms [25] have been devised for the deterministic version of the problem. In the case of statistical optimization, the algorithm must additionally be customized according to the input η value. We now proceed to describe two methods for the statistical version: one heuristic approach and the other a Simulated Annealing based search. We show how the customization of the algorithms to the required η value occurs in both cases.

6.1 List scheduling based heuristic

List scheduling algorithms are popular for solving multiprocessor scheduling problems [3]. They have been shown to perform well on common variants of scheduling problems [23].

Almost all list scheduling algorithms have the following structure. First, the algorithm computes a total ordering or list of tasks based on some priority metric. Tasks are then considered according to the list order and scheduled on available processors. The algorithm repeatedly executes the following two steps until a valid schedule is obtained:

- Select the next task to schedule, which is typically the task with highest priority and whose predecessors have completed execution.

- Allocate the selected task to a processor that allows the earliest start time for that task.

The algorithm progresses by allocating a task at each iteration and terminates when all tasks are scheduled. List scheduling algorithms differ in the rule used to pick the next task from the list when a processor is free [3]. A common choice of priority is the static level, the largest sum of execution times along any path from a task to any sink vertex in the graph [12]. Ties are broken based on the task processing times or the number of immediate successors [13]. More recent list scheduling algorithms have been based on dynamic levels, which are recomputed after each task has been allocated to a processor. One such algorithm is Dynamic List Scheduling [23] proposed by Sih and Lee, which computes a dynamic level for a (task, processor) pair as the difference between the static level and the earliest time that the task can start on that processor. This algorithm picks the best (task, processor) pair at each iteration and assigns the task to the processor.

When we consider statistical approaches, we need to define a priority metric for the tasks. The static and dynamic levels of a task are now distributions rather than deterministic values, since the execution time of each task varies across runs. We now show how to define these terms in the statistical setting. These definition will depend on the value of η, the required statistical guarantee.

Definition of Statistical Static Level (SSL).

The static level of a node is defined as the length of the longest path from the sink node to it in the task graph. The main motivation of using static levels is that at any given point in the scheduling heuristic, the static level of any unscheduled task, when added to the start time of the task, gives a valid lower bound to the optimal makespan. This is because the static level of a task added to its start time gives the longest path in the *unscheduled graph* through that node, while the optimal makespan is optimal makespan is the longest path on the *scheduled* graph with additional "ordering" edges. The greedy heuristic approach is to keep this lower bound as low as possible by scheduling the node with the highest static level first, giving it as low a start time as possible.

In the statistical case, the static level is a distribution. To obtain this distribution, we perform a Monte Carlo analysis on the original task graph (without any ordering edges). In each Monte Carlo iteration, we compute the static level with the corresponding deterministic execution time values for each task. We take the η^{th} percentile of this distribution to give us a single number for *statistical static level* that we use in our algorithm. We now justify this choice of statistical static level. Just as in the static case, we are interested in finding the task which will minimize the lower bound to the makespan. However, the definition of makespan is now the η^{th} percentile of the makespan distribution. We can see that, just as in the deterministic case, the η^{th} percentile of the static level distribution of the task, added to its start time, gives us the η^{th} percentile of the longest path through that node. It can easily be seen that this is a lower bound to the η^{th} percentile to the makespan: there cannot be a makespan value in the top η % that is lesser than a value outside the top η % of any longest path in the unscheduled graph. As before, we pick the node that minimizes this lower bound, or the node with the highest statistical static level.

We illustrate our definition of statistical static levels this by means of a simple example. Consider a simple task graph with four nodes: node A, node B, and the source and sink nodes. There are edges from the source node to nodes A and B, and edges from nodes A and B to the sink node. Node A has a deterministic execution time of 5, while node B can have an execution time of either 4 (with 80% probability) or 6 (with 20% probability). Both the source and sink nodes have zero execution time. The static level of nodes A and B are the same distributions as their respective execution times. The static level of the source node, or the longest path distribution will be the maximum of these two static levels. For this example, it is evident that the the distribution will be either 5 (with 80% probability) or 6 (with 20% probability). Now, if we are interested in a statistical guarantee $\eta > 80\%$, then the η^{th} percentile of the longest path has a value of 6. This, in turn, is a lower bound on the makespan. In this case, it is only node B that determines this lower bound. Any decrease in the execution time of node A has no effect on the lower bound on makespan. Our definition of statistical static levels reflects this: the level for node A is 5, while that of B is the η^{th} percentile of its static level distribution, which is 6. This allows node B to be scheduled earlier than A. However, if η is less than 80%, then the statistical static level of B becomes 4, less than that of A which always remains 5. Thus node A is scheduled earlier than B. This is also justified: we note that any delay in scheduling A will result in an increase to the lower bound of the makespan, which is currently the longest path through A (a value of 5). A delay in scheduling B would affect some values in the high end of the lower bound to makespan distribution, but not the η^{th} percentile itself.

Overall algorithm and the definition of Statistical Dynamic Levels.

The algorithm runs within a scheduling loop with as many iterations as the number of tasks in the graph. At each loop iteration, we maintain the current allocation A and schedule start time S for the tasks assigned up to the beginning of the loop. We also maintain the finish time F of all tasks on each processor so far. The schedule S and finish times F are probability distributions while A is deterministic. In each loop iteration, we consider only tasks that have not been assigned and all of whose predecessors have been assigned. For each such task v and for each processor p, we compute the earliest time $ST(v,p)$ that task v could start on the processor p. $ST(v,p)$ is merely the maximum of the latest finish time of all predecessors of v (plus communication time) and the time $F(p)$ when processor p becomes free. The finish times of the predecessor nodes as well as $F(p)$ are distributions, and hence we need to perform a statistical analysis to compute the maximum of these. We do this by means of an incremental Monte Carlo analysis that we explain later. The *statistical dynamic level* of (v,p) is defined as $SDL(v,p) = SSL(v) - \eta^{th}$ percentile of $ST(v,p)$. We then pick the (v,p) pair with highest $SDL(v,p)$ and assign $A(v) = p$. We also set the start time of task v $(S(v))$ to $ST(v,p)$ and update the time when processor p becomes free $(F(p))$ to $ST(v,p)+W(v,p)$, where $W(v,p)$ is the execution time distribution of task v on processor p. The computation of $F(p)$ also requires an incremental Monte Carlo step. We then move to the beginning of the iteration again and repeat until all nodes have been assigned.

We now explain our use of incremental Monte Carlo when computing $ST(v,p)$ and $F(p)$. We can think of the algorithm as building up a scheduled graph along with the "ordering" edges defining the schedule — but only containing a subset of nodes (the graph is being built in the order of dynamic levels). At each loop iteration of the algorithm, we have already performed a Monte Carlo analysis of the graph constructed so far – we have the Monte Carlo values of the S distribution for all predecessor tasks, as well as the Monte Carlo values for the F distribution for all processors. We are about to add a new node, with the predecessor edges as well as the "ordering" edge from the previous task on the same processor. To compute the new $ST(v,p)$ value, we only have to perform Monte Carlo simulations for the single node added (node v), and re-use the previously computed Monte Carlo samples for the predecessor nodes in the max computation. A similar situation arises for computing $F(p)$.

It should be noted that the above algorithm reduces to the deterministic Dynamic List Scheduling heuristic (DLS) if there are no variations in execution time. The definition of static levels used so far only works if all processors in the system are homogeneous. For a heterogeneous system, the average execution time of a task across different processors is used to compute the static level in [23]. We follow a similar scheme - we first compute for each task an average execution time distribution across different processors. We perform this averaging operation using a set of Monte Carlo simulations for each task independently. The result of this Monte Carlo simulation is the average execution time distributions that we then use in the succeeding steps for computing static levels.

6.2 Simulated Annealing

While heuristics have been found effective for certain multiprocessor scheduling problems, they are often difficult to tune for specialized problem instances. They may be additional constraints imposed on the problem in the form of topology constraints, constraints on task grouping and so on that are not easily handled by heuristics. A technique that has found success in exploring the complex solution space in practical scheduling problems is a Simulated Annealing (SA) based search [21][15]. SA is a global optimization algorithm that does not get stuck at local minima. The method is flexible and can incorporate a variety of objectives and parameters.

Simulated Annealing can be described as a probabilistic non-greedy algorithm that adapts the Markov chain Monte Carlo method for global optimization [14]. The idea of Simulated Annealing is well known, and so we do not go into many details. The algorithm starts with an initial state s_0, and a termination parameter t_∞. The algorithm goes through a number of iterations, each of which updates the current state s_i and a control parameter t_i, called temperature. The temperature determines how frequently inferior moves are accepted. The temperature is typically highest at the first iteration and is gradually reduced according to an "annealing schedule". At a given temperature, a random transition is proposed to a new state according to a $Move$ function. The $Cost$ function measures the quality of a state as a real number. After each move, the algorithm computes the difference Δ in the $Cost$ function between the current state s_i and the new state. The proposed transition is accepted if the new state improves the $Cost$ function; otherwise it is accepted with a probability that is dependent on Δ and the temperature t_i. The allowance for such "uphill" moves to inferior states saves the algorithm from becoming stuck at local minima and helps it cover a broad region of the search space. As the temperature decreases, the probability of uphill moves is lowered. The algorithm terminates when the temperature t_i decreases below t_∞.

We now discuss how simulated annealing can be used for statistical scheduling. The objective of the simulated annealing algorithm is to compute a valid allocation A and schedule S that yields a makespan distribution with the minimum η'th percentile for a given η. The inputs to the scheduling problem are the task dependence graph $G = (V, E)$, the multiprocessor architecture model P, and the task execution times $W(v)$ for all tasks in V. The task execution times are specified by means of a probability distribution table.

The key to a successful application of simulated annealing for the scheduling problem lies in the judicious selection and tuning of the $Cost$, $Move$, $Prob$ and $Temp$ functions. We found the $Prob$ and $Temp$ functions as defined in [15] to give us good scheduling performance. We discuss our choices for the $Move$ and $Cost$ functions below.

The annealing search, or the *optimization*, occurs over the set of all possible valid allocations and schedules. Every state is associated with a valid allocation and a valid schedule. The valid schedule at each state imposes a total order on the set of all tasks allocated to each processor (according to constraint (b) in Section 3.1). The $Move$ function randomly selects a task and a processor, and moves the task from its current processor to the randomly chosen one. $Move$ then randomly selects a position in the total order of the tasks in the new processor for the new task. As not all positions are acceptable due to ordering constraints, we check for the validity of the move and undo it if necessary. The global schedule needs to be recomputed after the relocation.

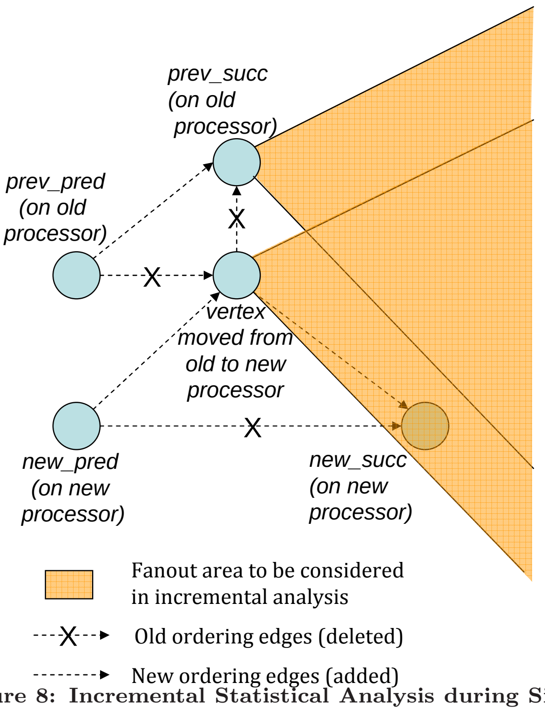

prev_succ (on old processor)
prev_pred (on old processor)
vertex moved from old to new processor
new_pred (on new processor)
new_succ (on new processor)
Fanout area to be considered in incremental analysis
Old ordering edges (deleted)
New ordering edges (added)

Figure 8: Incremental Statistical Analysis during Simulated Annealing.

Use of Incremental Timing Analysis.

After we obtain the new schedule, we need to compute the cost for the new state. An obvious choice for the $Cost$ function is the η'th percentile of the makespan distribution of the schedule. The $Cost$ function is the *analysis* function of Section 4.3. However, we do not need to perform the full Monte Carlo analysis because of the way in which the $Move$ function picks the next state.

Since the $Move$ function only changes one task to a different processor, the only change in the graph structure is involved with the ordering edges of the task that is moved. Before the move, the task had two ordering edges associated with it - one from its preceding task in the processor and one from its successor task. After the move, the task will also have two ordering edges associated with its successor and predecessor in the new processor. Figure 8 shows how the ordering edges change with a single transition. There are a total of six edges that change due to the transition. First, the edges from its predecessor and successor in the old processor are removed. Then an ordering edge is added from the predecessor to the successor of the task in the old processor. Third, the ordering edges to the predecessor and successor of the task in the new processor are added. Finally, the ordering edge between the predecessor and successor of the task in the new processor is removed.

The fanout of the six edges modified overlap significantly. In particular, the fanout of the successor task in the old schedule and the task that has been relocated are the only two tasks whose fanouts have to be considered. The incremental timing analysis described in Section 5 is used for this incremental analysis. We have found through experiments that we reduce the number of nodes to be considered by a factor of 2-3 on the average.

The simulated annealing algorithm can easily be used for the deterministic scheduling problem where each task's execution time is taken to the worst-case by changing the *Cost* function to the one in Section 4.4.

7. RESULTS

In this section, we show the results of applying the two statistical optimization approaches to the scheduling problem with execution time variations. We compare the makespans obtained by the two methods for statistical models at different percentiles with the result of deterministic simulated annealing for worst-case execution times. If the deterministic simulated annealing result was worse than the result of a deterministic Dynamic List Scheduling (DLS) [23] heuristic, then we used the heuristic value for comparison. All algorithms were run on a Pentium 2.4 GHz processor with 1 GB RAM running Linux.

Two benchmark sets were used in our experiments. The first benchmark set consisted of two task graph instances derived from practical applications from the multimedia domain: H.264 video decoding and MPEG-2 video encoding. The task graph that corresponds to decoding H.264 video streams was obtained from [5]. We schedule a frame at a time, and each task corresponds to a macro-block within the frame. The task execution times were profiled for a set of 1000 runs on a 2.4 GHz Pentium machine. The variations reflect the effect of cache misses on P-frames that access global frame buffer memory, as well as due to differing amounts of residue across different input streams. [5]. The code and task graph corresponding to MPEG-2 encoding was obtained from [7]. Task execution time variations were profiled for different frames. The second benchmark set was a collection of random task graph instances proposed by Davidović et al. [6]. These problems were designed to be unbiased towards any particular solver approach and are reportedly harder than other existing benchmarks for scheduling task dependence graphs. For the second set of benchmarks, we assumed that the distributions of execution times was normal. The mean execution times were taken from the benchmarks. The standard deviations were chosen randomly in the range [0 - 0.7*mean] such that the average ratio of standard deviation to the mean of each task is 0.35. We chose a value of 0.35 as this was the average ratio for the two practical applications.

Table 1 reports the results of the scheduling approaches for H.264 decoding and MPEG-2 decoding applications on 4, 6 and 8 processor systems. All processor systems were divided into two sets of processors, communication between which was about a factor of 4 more expensive than within the set (this models locality of processors on an on-chip multiprocessor network). Column 2 gives the number of tasks in the task dependence graph. Column 3 shows the number of processors that the graph is scheduled on. The subsequent columns report the makespan computed by the deterministic worst-case algorithm, the statistical list-scheduling heuristic and the statistical simulated annealing algorithms at percentiles of η = 99%, 95% and 90%. The reason the deterministic timing value varies across different percentiles is that we do not use the actual makespan computed by the worst-case analysis, which can be significantly higher than the statistical makespan. Instead, we only use the allocation and the "ordering" of tasks within each processor in the table, and perform a Monte Carlo analysis to obtain the makespan distribution for the deterministic worst-case schedule. We then read out the makespans corresponding to the required guarantee. This provides a fairer comparison than using the worst-case makespan values. The final column is the time taken for the statistical simulated annealing run. All heuristic runs completed within a minute.

# Tasks	Edge Density	Statistical improvement in % (η=)					
		99.9%		95%		90%	
		Heur.	SA	Heur.	SA	Heur.	SA
52	10	8.8	14.1	17.8	26.9	22.1	33.4
52	30	9.9	16.2	16.1	24.7	18.5	24.4
52	50	7.6	14.3	20.7	26.1	25.7	32.7
52	70	8.7	12.4	18.3	30.9	15.1	37.2
52	90	10.4	16.4	14.4	20.1	19.6	26.3
Avg. % diff. from worst-case		**9.1%**	**14.7%**	**17.5%**	**25.7%**	**20.2%**	**30.8%**
102	10	13.2	20.5	17.3	29.7	20.1	34.1
102	30	10.4	14.1	16.1	26.9	23.8	33.4
102	50	8.8	13.1	14.2	19.9	16.2	27.9
102	70	12.5	14.1	19.2	26.9	18.5	33.4
102	90	17.6	24.7	21.8	33.1	24.6	41.8
Avg. % diff. from worst-case		**12.5%**	**17.3%**	**17.7%**	**27.3%**	**20.6%**	**34.1%**

Table 2: Average percentage makespan difference between deterministic worst-case and statistical scheduling at different percentiles for random task graph instances (source: [6]) on 4, 6 and 8 processors.

Table 2 shows the results of the scheduling approaches on the benchmark with randomly generated task graphs. The benchmark instances are classified by the number of tasks and edge density (the percentage ratio of the number of edges in the task graph to the maximum possible number of edges). Columns 3 through 8 report the percentage difference between the deterministic worst case makespan and the statistical heuristic and simulated annealing makespan computed at percentiles of η=99.9%, 95% and 90% respectively. Each row is an average over the percentages for 4, 6 and 8 processors. All runs completed within 30 minutes. As before, the timing values are obtained by Monte Carlo runs on the schedule.

We observe from Table 1 that the deterministic schedule typically gives a schedule that is about 10-20% away from the simulated annealing schedule at η = 99.9%, and around 10% away from the heuristic schedule at the same η. This is also corroborated by the results in Table 2. This is because the deterministic scheduling method incorrectly compares (Sec. 4.5) different schedules, and hence yields a different and less optimal schedule than the statistical schedule. Further, as we decrease the required confidence interval to η = 95% and 90%, the deterministic schedule remains the same – we merely read off a different percentile from the Monte

Bench-mark	# Tasks	# Procs.	Statistical Guarantee $\eta =$									SA Opt. Time (s)
			99%			95%			90%			
			Worst-case	Stat. Heur.	Stat. SA	Worst-case	Stat. Heur.	Stat. SA	Worst-case	Stat. Heur.	Stat. SA	
H.264	182	4	2418.8	2160.1	2031.4	2397.7	2131.9	2005.0	2383.3	2042.7	1921.6	641
		6	1770.4	1595.5	1528.3	1751.4	1545.4	1422.1	1721.4	1455.1	1326.2	923
		8	1390.9	1287.1	1210.5	1376.2	1241.3	1119.4	1367.9	1197.3	997.8	1076
Avg. % diff. from det. worst-case				**10.3%**	**16.6%**		**12.2%**	**21.9%**		**16.4%**	**30.3%**	
MPEG-2	76	4	1247.2	1159.1	1121.6	1229.6	1094.9	1029.8	1216.8	1117.4	1040.4	399
		6	1190.8	1131.9	1084.9	1171.7	1022.4	965.0	1167.4	1000.3	920.1	457
		8	1179.5	1107.5	1070.4	1153.2	1080.8	1015.9	1146.3	980.6	903.6	621
Avg. % diff. from det. worst-case				**6.4%**	**10.4%**		**11.2%**	**18.1%**		**14.2%**	**23.5%**	

Table 1: Makespan results for H.264 decoding and MPEG-2 encoding comparing deterministic worst-case scheduling with statistical list-scheduling heuristic and simulated annealing (SA) algorithms at different percentiles

Carlo simulation. On the other hand, both statistical algorithms are customized to the particular value of η and actually change the allocation and ordering. The difference between the deterministic and statistical simulated annealing increases to about 25-30%, and the difference to the heuristic changes to about 20%. We also note from Table 2 that statistical optimization gives us more improvement on larger task graphs. This is because large task graphs tend to have many paths with differing variance in path delays. It is more likely for a large task graph to have some near-optimal paths with high variance. The deterministic worst-case estimate overestimates the impact of these paths, and could optimize them at the expense of lower variance paths, leading to suboptimal results. On the whole, the worst-case schedule tends to have a very low variance, which is also borne out in the tables, as the difference between the 99% and 90% percentiles for our applications is less than 3%.

8. SUMMARY

In this paper, we presented two statistical optimization approaches to schedule task dependence graphs with variations in execution time onto multiprocessors. We optimize for makespan for a required percentile. Static approaches based on worst-case estimates are one way to solve the problem, but are inaccurate. We propose statistical analysis to give makespan estimates that are reliably close to Monte-Carlo simulations. We use this analysis in a heuristic and a simulated annealing based optimization approach and demonstrate a 25-30% improvement in makespan over approaches based on worst-case estimates on soft-real time applications.

In the future, we intend to investigate other optimization methods based on constraint programming, which can give us optimal solutions to the scheduling problem. We also plan to integrate our solution methods into a practical design space exploration tool for multiprocessor platforms.

9. REFERENCES

[1] J. C. Beck and N. Wilson. Job Shop Scheduling with Probabilistic Durations. 2004.

[2] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer, 1997.

[3] C. Chekuri. *Approximation Algorithms for Scheduling Problems.* PhD thesis, Computer Science Department, Stanford University, Aug 1998. CS-TR-98-1611.

[4] W. Y. Chen, S. A. Mahlke, N. J. Warter, S. Anik, and W. mei W. Hwu. Profile-assisted instruction scheduling. *Int. J. Parallel Program.*, 22(2):151–181, 1994.

[5] J. Chong, N. R. Satish, B. Catanzaro, K. Ravindran, and K. Keutzer. Efficient Parallelization of H.264 Decoding with Macro Block Level Scheduling. In *2007 Intl. Conference on Multimedia and Expo*, pages 1874–1877, July 2007.

[6] T. Davidović and T. G. Crainic. Benchmark-Problem Instances for Static Scheduling of Task Graphs with Communication Delays on Homogeneous Multiprocessor Systems. *Computers and OR*, 33:2155–2177, 2006.

[7] M. Drake, W. Thies, and S. Amarasinghe. MPEG Coding in StreamIt. http://www.cag.csail.mit.edu/streamit/mpeg/.

[8] C. V. et al. First-Order Incremental Block-Based Statistical Timing Analysis. *IEEE Trans. on Computer-Aided Design*, 25(10):2170–2180, October 2006.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[10] H. Gautama and A. J. C. van Gemund. Static Performance prefiction of data-dependent programs. In *2nd Intl. Workshop on Software and Performance*, pages 216–226, Sep 2000.

[11] M. Gries. Methods for Evaluating and Covering the Design Space during Early Design Development. *Integr. VLSI J.*, 38(2):131–183, 2004.

[12] T. Hu. Parallel Sequencing and Assembly Line Problems. *Oper. Res.*, 19(6):841–848, Nov 1961.

[13] H. Kasahara and S. Narita. Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing. *IEEE Trans. on Computers*, C-33(11), Nov 1984.

[14] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):498–516, May 1983.

[15] P. Koch. *Strategies for Realistic and Efficient Static Scheduling of Data Ind. Algorithms onto Multiple Digital Signal Processors.* PhD thesis, Aalborg University, 1995.

[16] Y.-K. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[17] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987.

[18] M. Mani and M. Orshansky. A New Statistical Optimization Algorithm for Gate Sizing. In *Proc. of the IEEE Intl. Conf. on Comp. Design*, pages 272–277, 2004.

[19] S. Manolache, P. Eles, and Z. Peng. *Real-Time Applications with Stochastic Task Execution Times Analysis and Optimisation.* Springer Netherlands, 2007.

[20] P. Moge and A. Kalavade. A Tool for Performance Estimation of Networked Embedded End-Systems. In *DAC '98: 35th conf. on Design Automation*, pages 257–262, 1998.

[21] H. Orsila, T. Kangas, E. Salminen, and T. D. Hamalainen. Parameterizing Simulated Annealing for Distributing Task Graphs on Multiprocessor Socs. In *Proc. of the Intl. Symposium on System-on-chip*, pages 1–4, Nov 2006.

[22] K. Ravindran. *Task Allocation and Scheduling of Concurrent Applications to Multiprocessor Systems.* PhD thesis, University of California, Berkeley, Nov 2007.

[23] G. C. Sih and E. A. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Trans. Parallel Distrib. Syst.*, 4(2):175–187, 1993.

[24] A. Srivastava, D. Sylvester, and D. Blaauw. Statistical optimization of leakage power considering process variations using dual-Vth and sizing. In *DAC '04: Proc. of the 41st conf. on Design automation*, pages 773–778, 2004.

[25] L. Thiele. Resource Constrained Scheduling of Uniform Algorithms. *VLSI Signal Proc.*, 10(3):295–310, Aug 1995.

[26] A. J. C. van Gemund. *Performance Modeling of Parallel Systems.* PhD thesis, Delft University of Technology, 1996.