# A Period Graph Throughput Estimator for Multiprocessor Systems[1]

*Neal K. Bambha and Shuvra S. Bhattacharyya*

*Department of Electrical and Computer Engineering, and*
*Institute for Advanced Computer Studies*
*University of Maryland, College Park*

## Abstract

*A critical challenge in synthesis techniques for iterative applications is the efficient analysis of performance in the presence of communication resource contention. To address this challenge, we introduce the concept of the period graph. The period graph is constructed from the output of a simulation of the system, with idle states included in the graph, and its maximum cycle mean is used to estimate overall system throughput. As an example of the utility of the period graph, we demonstrate its use in a joint power/performance optimization solution that uses either a nested genetic algorithm, or a simulated annealing algorithm. We analyze the fidelity of this estimator, and quantify the speedup and optimization accuracy obtained compared to simulation.*

## 1 Introduction

In many practical multiprocessor systems, there is contention for one or more shared communication resources. One example of this is a shared bus. A processor must first gain access to the bus before it can execute an interprocessor communication (*IPC*) operation. One consequence of this contention is that under self-timed, iterative execution, there is no known method for deriving an analytical expression for the throughput of the system [19], and thus, simulation is required to get a clear picture of application performance. However, simulation is computationally very expensive, and it is highly undesirable to perform simulation inside the innermost optimization loop during synthesis. To avoid such a simulation, an accurate and efficient estimator for throughput is required. This paper presents an efficient estimator for the throughput of these systems. Our work is in the context of self-timed execution of iterative dataflow specifications, which is an efficient and popular design methodology in the domain of digital signal processing (DSP) [13]. An iterative dataflow specification consists of a dataflow representation of the body of a loop that is to be iterated a large or indefinite number of times (e.g., across a vast stream of speech samples). In self-timed execu-

1. Technical Report UMIACS-TR-2000-49, Institute for Advanced Computer Studies, University of Maryland at College Park, 2000. This research was sponsored in part by the US National Science Foundation under Grant #9734273, and the US Army Research Laboratory under Contract #DAAL01-98-K-0075 and the MICRA program.

tion, the assignment of tasks (dataflow graph nodes) to processors, and the execution ordering of tasks on each processor are determined at compile-time, and at run-time, processors synchronize with one another only based on inter-processor communication requirements, and do not necessarily synchronize at the end of each loop iteration.

In this paper, we assume that a deterministic protocol is used to arbitrate contention for communication resources. We assume that a schedule has already been computed so the order of the tasks on the processors is known, and that we are adjusting some task parameters that vary the task execution times in order to perform an optimization of the system. We assume that reasonably accurate estimates are available for the task execution times, and for the variation of execution times with parameter changes. Later in the paper, we specifically address the problem of finding an optimum set of supply voltages for the processors in order to reduce power while satisfying a throughput constraint.

## 2 Previous work

The estimates for task execution times can be obtained through several methods. The most straightforward is for the programmer to provide them while developing a library of primitive blocks, as is done in the Ptolemy system [3]. Analytical techniques also exist. Li and Malik [14] have proposed algorithms for estimating the execution time of embedded software in an efficient manner. Much work has been done on scheduling and binding methods for high level synthesis [16][8][9][7]. These techniques attempt to optimize the schedule *makespan*, which is a suitable performance metric for non-iterative applications or fully-static implementations, but is not ideally suited to the iterative, self-timed context that we address in this paper. Supply voltage reduction has been used for some time in memories and consumer electronics [15]. Chandrakasan et al. [5][6] have presented a method based on reduced voltage level operation combined with architectural-level parallelism, showing that the throughput can be maintained while reducing power. Tiwari et al. [21] presented a technique for estimating the power given a set of software instructions. This technique can be used in conjunction with the approaches proposed in this paper to obtain more accurate or automated estimates for the power consumption of the tasks in period graph model.

The period graph model is inspired by the *synchronization graph* model [19] for self-time multiprocessor systems. The synchronization graph has proven useful for a variety of techniques for minimizing synchronization overhead, allocating interprocessor communication buffers, and scheduling communication operations [11, 19]. The period graph concept differs from the synchronization graph model in that it explicitly models steady-state behavior under communication resource contention, which is not accounted for in synchronization graphs.

A preliminary, partial summary of this paper has been published in [2].

## 3 Period Graph

If contention is resolved deterministically, and execution times are constant, then self-timed evolution may lead to an initial transient state, but the execution will eventually become periodic. This holds because the multiprocessor may be modeled as a finite-state system, and thus, aperiodic behavior — which implies the presence of infinitely many distinct states — cannot hold. In DSP systems, although execution times are not always constant, or known precisely, they typically adhere closely to their respective estimates with high frequency. Under such conditions, the periodic execution pattern obtained from the estimated execution times provides an estimate of overall system throughput based on the task-level estimates. Due to the largely deterministic nature of DSP applications, such system-level performance analysis, and optimization based on task-level estimates is common practice in the DSP design community [13].

For self-timed systems, when we apply execution time estimates to estimate overall throughput, it is necessary to simulate (using the execution time estimates) past the transient state until a periodic execution pattern (steady state) emerges. Unfortunately, the duration of the transient may be exponential in the size of the application specification [19], and this makes simulation-intensive, iterative synthesis approaches highly unattractive.

The objective in this paper is to greatly reduce the rate at which simulation must be carried out during iterative synthesis through the use of a novel *period graph* model. Given an assignment $\nu$ of task execution times, and a self-timed schedule, the associated period graph is constructed from the periodic, steady-state pattern of the resulting simulation. The maximum cycle mean (*MCM*) of the period graph (with certain adjustments) is then used as a computationally-efficient means of estimating the iteration period (the reciprocal of the throughput) as changes are explored within a neighborhood of $\nu$. In this context, the MCM is the maximum over all directed cycles of the sum of the task execution times divided by the sum of the edge delays. The MCM can be computed in low polynomial time [12].

The first step in the construction of the period graph is the identification of the period from the simulator output. This can be performed by tracing backward through the simulation and searching for the latest intermediate time instant $t_a$ at which the *system state* $S(t_a)$ equals the state $S(t_f)$ obtained at the end of the simulation (here, $t_f$ denotes the simulation time limit). If no match is found, then the end of the first period exceeds $t_f$, and thus, the simulation needs to be extended beyond $t_f$. Otherwise, the region (Gantt chart) that spans the interval $[t_a, t_f]$ constitutes a (minimal) period of the simulated steady state.

Here, the system state $S(t)$ contains the execution state of each processor, which is either “idle” or representable by an ordered pair $(A, \tau)$, where $A$ is the task being executed at time $t$, and $\tau$ denotes the time remaining until the current invocation of $A$ is completed. The state $S(t)$ also contains the current buffer sizes of all IPC buffers, as well as any information (e.g., request queue status) that is used by the protocol for resolution of communication contention. Our approach to efficiently determining this period is as follows:

- Perform a simulation of the schedule for some time $T_{sim}$. Define a constant $C$, which is an initial estimate for the number of complete cycles (invocations) of the graph that must be simulated in order to find a period. This constant represents the length of the initial transient, before the output becomes periodic. If this initial estimate is too low, it will be increased during the algorithm. Let $N$ be the number of processors, and let $n_j$ be the number of tasks scheduled on processor $j$, where $j \in [1,N]$. Tasks include IPC tasks as well as computational tasks. Label these tasks $V_{1_j}, V_{2_j} \ldots V_{n_j}$. We consider the case where the system executes these tasks infinitely. The *invocation number* of a task is defined as the number of times a given task has executed, and is denoted with a superscript. For example, $V^{b(j)}{}_{a(j)}$ denotes the $b_{th}$ invocation of task $a$ on processor $j$. Define a simulation array for each processor $\text{Sim}_j[i]$ where $i \in [1,M_j]$ and $M_j$ is the number of tasks on processor $j$ that were output by the simulator. The elements of the simulation array are the tasks, and are ordered by reverse start time, so that $\text{Start}(\text{Sim}_j[i]) > \text{Start}(\text{Sim}_j[i+1])$.

- Create two *idle vectors* of length $n_j$ for each processor spanning one invocation. Label the first idle vector $\mathrm{Idle1}^1{}_j[k]$ where $k \in [1,n_j]$. Label the second idle vector $\mathrm{Idle2}^1{}_j[k]$.
- Examine the *IPC buffer vector* at some fixed point of each idle vector. The IPC buffer vector consists of the numbers of tokens queued on all the IPC edges of the graph enumerated in some order. The IPC buffer vector must be output by the simulator at least once every graph iteration. For example, the simulator could output an IPC buffer vector for each processor every time the processor executes the first task scheduled on it. In this way, each idle vector would be associated with one IPC buffer vector. Label these vectors $\mathrm{IPCBuf1}_j[q]$ and $\mathrm{IPCBuf2}_j[q]$ where $q \in [1,E]$ and $E$ is the number of edges in the IPC graph. The IPC buffer vector represents the state of the communication buffers in the system. Let $\mathrm{Tokens}(e, t)$ be the number of data tokens on edge $e$ at time $t$. Let $\mathrm{TaskNum}_j(t)$ be the number of the node that is executing on processor $j$ at time $t$. Pseudo-code for constructing the period is shown in Figure 2:

Our experience suggests that in practice, most graphs have periods spanning only a few invocations, so the above procedure for finding the period is efficient. For a system with a period that spans $N$ invocations and with at most $L$ tasks per processor, this method requires $LN(N+1)$ comparisons.

Figure 1(a) illustrates an *application graph* (a dataflow specification of an application) along with a self-timed sched-

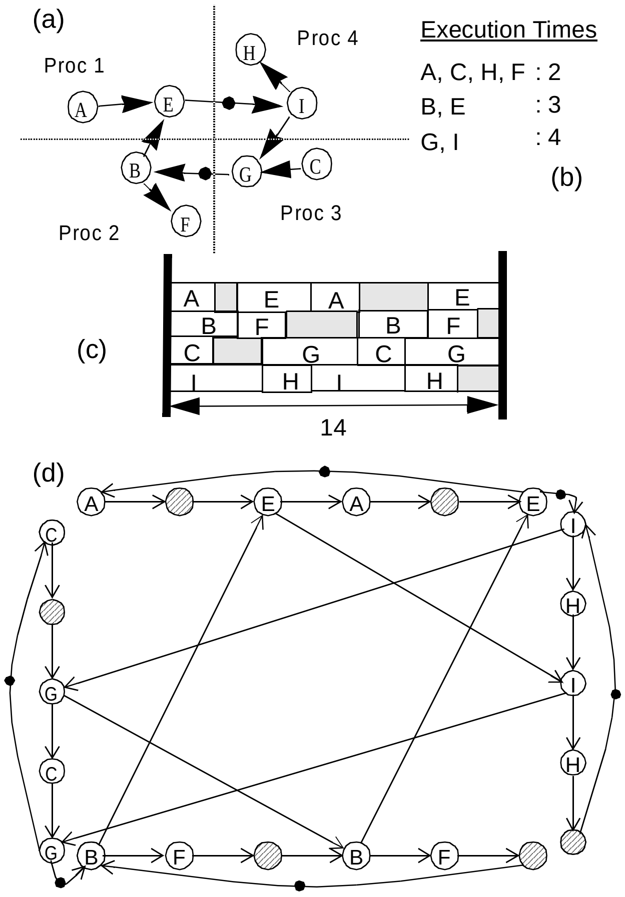


Figure 1An illustration of the period graph model.

```
Procedure CalculatePeriod

estimate initial integer C and initial simulation time $T_{\text{sim}}$
int $\text{minc} = 0$
while ( $\text{minc} < C$ )
    Increment $T_{\text{sim}}$
    Simulate for $T_{\text{sim}}$

    $\text{minc} = \text{minimum over all j}\left(\left\lfloor \frac{M_j}{n_j} \right\rfloor\right)$

endwhile


$t = 0$
for ($t = 0 ; t < T_{\text{sim}} ; t{+}{+}$)
    for ($j = 1 \ldots N$)
        $a_j = \text{TaskNum}_j(t)$
        $\text{invocation}_j[a]{+}{+}$
        $b_j = \text{invocation}_j[a]$
        if ($\text{TaskNum}_j(t) > \text{TaskNum}_j(t-1)$)
                $\text{Sim1}_j[i] = V^{b(j)}{}_{a(j)}$
        endif
    endfor
endfor

$\text{span} = 0$

Repeat
    $\text{span}{+}{+}$
    for ($k = 1 \ldots \text{span*n}_1$)
        for($j = 1 \ldots N$)
                if($\text{span*}n_j > M_j$)
                        error("Increase C and start over")
                endif
            $\text{Idle}_j^1[k] = \text{Finish}(\text{Sim}_j[k]) - \text{Start}(\text{Sim}_j[k+1])$
            $\text{Idle}_j^2[k] = \text{Finish}(\text{Sim}_j[\text{span*}n_j + k]) - \text{Start}(\text{Sim}_j[\text{span*}n_j + 1 + k])$
        endfor
        for ($q = 1 \ldots E$)
            $\text{IPCBuf1}[q] = \text{Tokens}(q, \text{Start}(\text{Sim}_1[1]))$
            $\text{IPCBuf2}[q] = \text{Tokens}(q, \text{Start}(\text{Sim}_1[\text{span*}n_1 + 1]))$
        endfor
  endfor
Until ($\prod_j (\text{Idle}_j^1 \equiv \text{Idle}_j^2) = 1$) and ($\text{IPCBuf1} = \text{IPCBuf2}$)
```

Figure 2. Pseudocode specification for period calculation.

ule; Figure 1(c) shows the periodic steady state that results from the schedule of Figure 1(a) and the execution time estimates shown in Figure 1(b); and Figure 1(d) shows the resulting period graph. The nodes in Figure 1(d) that contain diagonal stripes correspond to idle time ranges in the period, and solid black circles on edges represent delays, which model inter-iteration dependencies. Note that the steady state period may span multiple graph iterations (2 in this example), and in the period graph, this translates to multiple instances of each application graph task.

For clarity in this illustration, we have assumed negligible latency associated with IPC. As described below, non-negligible IPC costs can easily be accommodated in the period graph model by introducing *send* and *receive* tasks at appropriate points.

As illustrated in Figure 1, the period graph consists of all the tasks comprising the period that was detected, with the idle time ranges between tasks (including those that are caused by communication contention) also treated as nodes in the graph. The nodes are connected by edges in the order that they appear in the period. An edge is placed from the last node in the period for each processor to the first node in the period. This edge is given a delay value of one (to model the associated transition between period iterations), while all of the other intraprocessor edges have delay values of zero. This is done for all the processors in the system. Our model utilizes *send* and *receive* nodes for IPC. For each IPC point, a send node is placed on the processor that is sending data, and a corresponding receive node is placed on the processor that will receive the data. The period graph is completed by adding an edge from each send node to its corresponding receive node.

## 4 Fidelity of the estimator

We calculate the fidelity of the period graph estimator as the task execution times are varied. Here, we use the example of varying the processor voltages in order to change the task execution times. When the voltage on a processor is varied, the execution time of a computational task varies according to

$$delay = k \cdot \frac{V_{dd}}{(V_{dd} - V_t)^2} . \tag{1}$$

where $V_{dd}$ is the supply voltage, $V_t$ is the threshold voltage, and $k$ is a constant [6]. We use a value of $0.8 volts$ for the threshold voltage. The execution time $pe_i$ of each of these states in the original (non-scaled) period graph is referenced to a voltage $V_{ref}$. The change in execution time of each computation node is found by taking the derivative:

$$\Delta pe_i = pe_i \left( \frac{V_{sc}}{V_{ref}} \left[ \frac{1 - \frac{V_t}{V_{sc}}}{1 - \frac{V_t}{V_{ref}}} \right]^2 - 1 \right), \tag{2}$$

where $V_{sc}$ is the new voltage. It is not obvious, however, how one should adjust the idle times in the period graph. We separate the idle nodes into two sets: *contention idles* and *data idles*. When a node has the necessary data to execute (the necessary data has already been produced), but is idle waiting for access to the bus, the associated idle node is classified as a contention idle. When a node is idle waiting for its predecessors' data, the associated idle node is classified as a data idle. By experimenting with a large number of application graphs, we found that we could capture the effects of contention and obtain the best fidelity by zeroing out the data idles and leaving the contention idles constant as the computation idles are scaled. Using these rules, the fidelity is calculated as follows:

- Given an application graph, construct a valid schedule. We used the dynamic level scheduling algorithm given in [17]. Next, construct the period graph as discussed earlier. Generate $N$ voltage vectors (assignments of voltages to the processors in the target architecture). For each voltage vector, perform a simulation to determine the throughput, with the execution times of the tasks on each processor given by (1) according to the voltage on the processor. Also, obtain an estimate for the throughput by calculating the MCM of the voltage-scaled period graph, in which the execution times of the computation nodes are given by (1), and the execution times of the idle nodes are as explained above.
- Calculate the fidelity according to:

$$Fidelity = \frac{2}{N(N-1)}\left(\sum_{i=1}^{N-1}\sum_{j=i+1}^{N} f_{ij}\right) , \tag{3}$$

where

$$f_{ij} = \begin{cases} 1 \text{ if } sign(S_i - S_j) = sign(M_i - M_j) \\ 0 \text{ otherwise} \end{cases} ; \tag{4}$$

$$sign(x) = \begin{cases} (-1) \text{ if } (x < 0) \\ 0 \text{ if } (x = 0) \\ 1 \text{ if } (x > 0) \end{cases} ; \tag{5}$$

the $S_i$ s denote the simulated throughput values; and the $M_i$ s are the corresponding estimates from the period graph.

Figure 3 plots $Fidelity$ for a six-processor system in which the voltage on the individual processors can vary between plus or minus five percent. The x-axis represents the sum of the absolute values of the voltage changes over all processors. Each point on the graph is a fidelity calculation for $N = 100$ voltage vectors. A value of one is a “perfect” fidelity. It can be seen that in the range shown, the fidelity is always greater than 0.65. It is also important that the estimator have a small error at each point. Figure 4 plots

$$\sum_{i=1}^{N} [(S_i - M_i)/S_i] . \tag{6}$$

It can be seen that the error increases as the voltage vector moves away from the reference point, and that the estimate is slightly biased. For the range shown in the graphs, where each processor voltage is changed by a maximum of fifteen percent, the error is less than four percent.

## 5 Using the Period Graph in a Joint Power/Performance Algorithm

An effective way to reduce power consumption of a processor core in CMOS technology is to lower the supply voltage level, which exploits the quadratic dependence of power on voltage [6]. Reducing the supply voltage also has the effect of decreasing the clock speed and increasing circuit delay. The circuit delay can be modeled by (). The power consumption is

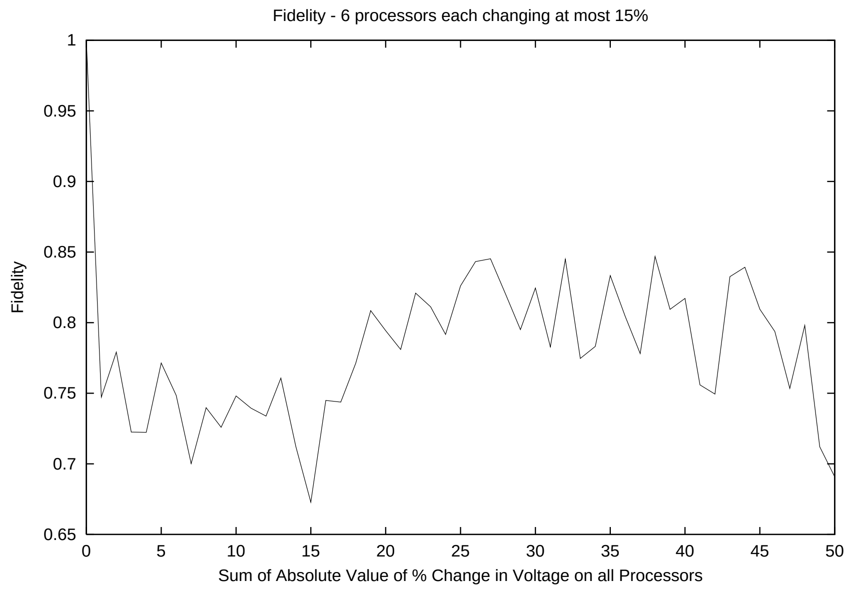


Figure 3. Plot of fidelity (equation 3) for six processor system vs. magnitude of voltage change on all processors

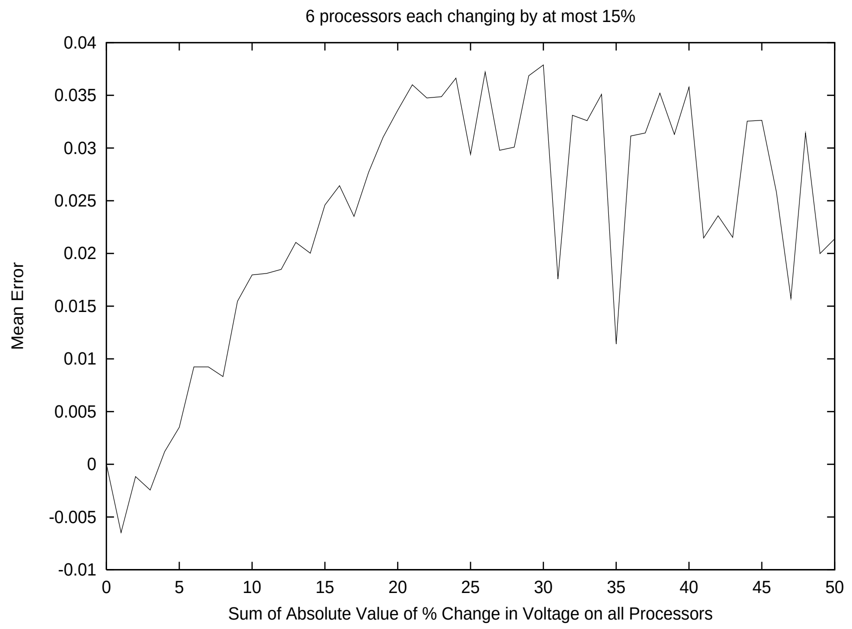


Figure 4. Plot of average error vs. voltage change on processors.

given by

$$P = \alpha C_L V_{dd}^2 f, \tag{7}$$

where $f$ is the clock frequency, $C_L$ is the load capacitance, and $\alpha$ is the switching activity [6]. To accommodate the possibility of putting processors in states of lower switching activity during idle periods, our model includes a parameter $\alpha_{idle}$ for the idle states, and a parameter $\alpha_{non-idle}$ for the computational tasks, where $\alpha_{idle} \le \alpha_{non-idle}$. A more detailed power analysis could assign a different $\alpha$ for each computational task if that data were available. A different power optimization technique, which can be used in conjunction with the voltage scaling technique presented here, utilizes a nearly complete processor shutdown during the idle periods [10][20]. In our model, this would correspond to $\alpha_{idle} = 0$. Our model for the power is the average energy consumption per graph iteration period. This corresponds in a typical DSP system to the average energy required to process one sample. Here, the energy of each node equals its power times its execution time.

In a system consisting of multiple processors, one has the ability to choose, within a certain range, the (fixed) operating voltage on each processor. This opens up an additional degree of freedom that can be exploited to minimize the system power consumption. By choosing a lower voltage of a processor that is executing tasks that are not on the critical path, the throughput can remain unchanged while the overall power consumption is reduced. In general, a combination of raising voltages on some processors while lowering others can yield the most attractive power/performance solution.

When applying voltage scaling to a multiprocessor system, the valid solution space is typically much too large to search by brute-force methods. In addition, since there is no general analytical formula for calculating the throughput of these systems in the presence of communication resource contention, each candidate solution must either be simulated or estimated using some heuristic.

## 6 Genetic algorithm formulation

To demonstrate the general utility of the period graph based performance estimation approach, we incorporated it into two significantly different probabilistic search techniques to derive two different algorithms for systematic voltage scaling. The first algorithm presented utilizes the framework of genetic algorithms (GAs) [1]. The specific GA explored here consists of an inner GA nested within an outer GA. The inner GA performs a local search around a point from the population of the outer GA, using the MCM of the period graph in its objective function as an estimate for the throughput. A period constraint $T_{\text{constraint}}$ is given as an input to the optimization problem, where the period is the reciprocal of the throughput. The objective function calculates the power consumption associated with each solution by calculating the total energy per period, as discussed earlier. If the period associated with a solution violates the period constraint $(T_{\text{solution}} > T_{\text{constraint}})$, the power consumption is multiplied by a large penalty factor $\exp(100(T_{\text{solution}} - T_{\text{constraint}}))$. The GA attempts to minimize this objective function.

In the outer loop, a population of $N_{\text{outer}}$ voltage vectors is generated. A simulation is run and a period graph constructed for each of these outer loop voltage vectors. For each of the outer loop voltage vectors, a new inner loop population is generated such that $|\text{Vouter}_i - \text{Vinner}_i| < \varepsilon$ for $i \in N_{\text{proc}}$, where $N_{\text{proc}}$ is the number of processors,

$\mathrm{Vouter}_i$ is the voltage on processor $i$ in the outer population, $\mathrm{Vinner}_i$ is the voltage on processor $i$ in the inner population, and $\varepsilon$ is a user-defined threshold. The inner population size is $N_{\mathrm{inner}}$. The inner GA then performs a local search using this population for a number of generations $\mathrm{Generations}_{\mathrm{inner}}$ in an attempt to find a locally optimal voltage vector. The inner GA uses the MCM of the period graph in its objective function. After an invocation of the inner GA is finished, one simulation is performed using the resulting voltage vector, and the actual throughput for this point is used to compute its fitness. The outer loop voltage vector is then replaced with this locally-optimized voltage vector for use in the next outer loop generation. The outer loop is run for a number of generations $\mathrm{Generations}_{\mathrm{outer}}$.

## 7 Simulated annealing algorithm

Simulated annealing is another well-known method for searching large design spaces. Using a standard simulated annealing package [4], we have implemented an alternative version of period-graph-based voltage scaling optimization. The objective function here is the same as for the genetic algorithm. The system is first simulated with an initial voltage vector $\mathrm{V}_j = \mathrm{LSV}_j$, and the period graph is built. In order to insure that the period graph will be a good enough estimator, a *resimulation threshold* $T$ is maintained. The difference between the current input $\mathrm{CV}_j$ to the objective function, and the voltage vector $\mathrm{LSV}_j$ corresponding to the simulation used to compute the current period graph, is calculated. If

$$\frac{1}{N}\sum_{i=1}^{N}\left|\frac{V_i - \mathrm{LSV}_i}{\mathrm{LSV}_i}\right| > T, \tag{8}$$

the graph is resimulated using $\mathrm{CV}_j$. The period graph is rebuilt, and $\mathrm{CV}_j \to \mathrm{LSV}_j$. For $T = 0$, the graph will be resimulated every time, and the period graph will offer no speed advantage. The larger the value of $T$, the less often the graph will be resimulated, and the faster the optimization algorithm will perform. However, when $T$ is too large, the fidelity of the period graph estimate will be unacceptably low and the quality of the final result will suffer. Based on our experiments with a number of graphs, the optimal value of $T$ is highly application-dependent, but a value of $T = 0.1$ (10%) generally gives good results.

## 8 Results

Figure 5 shows an example of the reduction in power resulting from the genetic optimization algorithm on the FFT3 application graph (Figure 10). The parameters of the GA were $N_{\mathrm{outer}} = N_{\mathrm{inner}} = 50$, $\mathrm{Generations}_{\mathrm{outer}} = 10$, $\mathrm{Generations}_{\mathrm{inner}} = 20$. The local search voltages were constrained to be within five percent of the corresponding outer loop voltages. The period constraint was calculated by simulating the system with all six processors operating at voltage $V_{ref}$. For this example, the system power consumption was reduced by 43%, while maintaining the original throughput. To evaluate the advantage of the period graph approach over using brute-force simulation, a second nested GA was implemented. This algorithm was identical to the algorithm discussed above, except that the inner loop did not use the period graph estimate for the throughput. Instead, each voltage vector was evaluated by simulation. This algorithm consumed 26 times more CPU time, and produced similar results, as shown in Figure 6.

Figure 7 summarizes the power reduction results for the simulated annealing algorithm applied to a fast Fourier

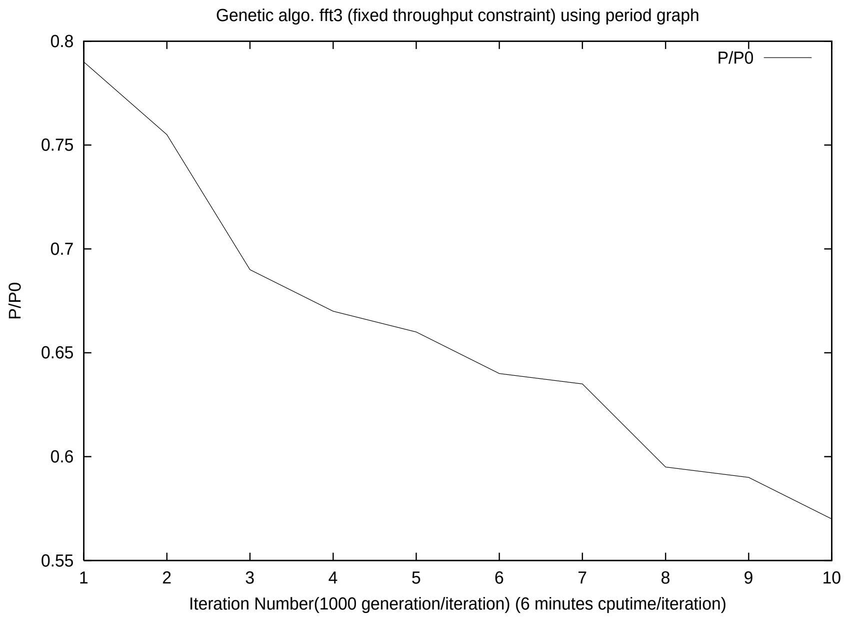


Figure 5. Plot of (*optimized power*)/(*initial power*) vs. genetic algorithm iteration using the period graph estimator.

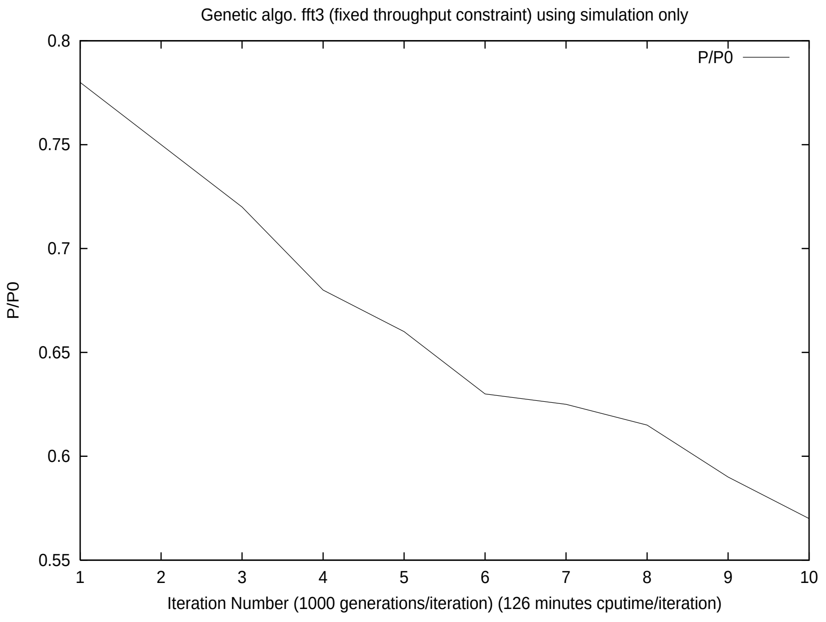


Figure 6. Plot of (*optimized power*)/(*initial power*) vs. genetic algorithm iteration using simulation only.

transform (FFT3) application graph, for different values of the resimulation threshold $T$. It can be seen that as $T$ is increased, the algorithm progresses more quickly. The simulated annealing algorithm begins with a ‘melting’ routine, where the temperature is increased until a phase change is detected. The initial flat part of the curves corresponds to the time spent in the melting routine. We have found that for values of $T$ above 20%, the period graph is not a good enough estimator and the algorithm does not converge.

Table 1 summarizes the power reduction for the simulated annealing algorithm for several additional applica-

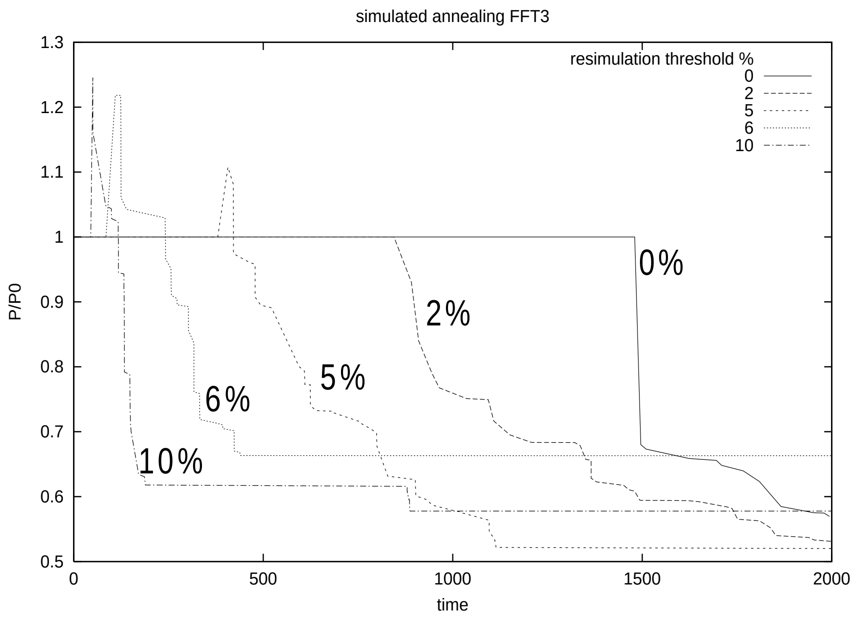


Figure 7. Plot of (*optimized power*)/(*initial power*) vs. time for simulated annealing algorithm on FFT3 application.

| **application** | **0** | **2%** | **5%** | **10%** | **25%** |
|---|---|---|---|---|---|
| fft1 (28) | 0.96 | 0.95 | 0.65 | 0.6 | 1 |
| fft2 (28) | 0.97 | 0.9 | 0.71 | 0.97 | 1 |
| fft3(28) | 1 | 0.77 | 0.59 | 0.59 | 1 |
| mus (20) | 0.89 | 0.71 | 0.67 | 0.82 | 1 |
| meas (12) | 0.77 | 0.73 | 0.81 | 0.82 | 1 |
| qmf (14) | 0.84 | 0.65 | 0.67 | 0.73 | 1 |
| rand1 (30) | 0.91 | 0.77 | 0.53 | 0.65 | 1 |
| rand2 (100) | 1 | 0.85 | 0.77 | 0.73 | 1 |
| rand3 (200) | 1 | 1 | 1 | 0.94 | 1 |

Table 1. power reduction for fixed computation time.

tions using different values of the resimulation threshold. At the start of the optimization, all processor voltages were set at 5 volts. The throughput at this point was used as the throughput constraint. In the table, the first three rows correspond to three different FFT implementations, *mus* refers to a music synthesis algorithm, *qmf* refers to a quadrature mirror filter bank, *meas* is a measurement application, and the last three rows correspond to graphs that were generated using Sih's algorithm for randomly generating application graphs [18]. The numbers in parentheses give the numbers of nodes in these applications. The optimization was performed for a *fixed time* of 30 minutes in each case. The optimum resimulation threshold was between 2% and 10% in all cases. For $T = 0.25$, the period graph is not a good estimator and none of the results returned during the optimization algorithm satisfied the throughput constraint. For the largest graph, the fixed simulation time was not long enough to make much improvement, but the best result occurred for $T = 0.1$, where the simulations are less frequent. Table 2 summarizes the power reduction for the genetic algorithm with and without using the period graph, with a fixed compile time of one hour.

## 9 Conclusion

This paper has explored a *period graph* model that enables efficient voltage scaling optimization for self-timed implementations of iterative applications. The period graph can be used as a computationally efficient estimator for the throughput in multiprocessor systems in which communication contention renders exact analysis too time-consuming. This model is especially useful in iterative synthesis techniques, such as those based on probabilistic search. Our paper has demonstrated effective voltage scaling techniques based on incorporating the period graph into genetic algorithm and simulated annealing formulations. Other optimizations, such as exploiting memory/speed trade-offs of the individual tasks, are also possible. These may be more appropriate to the genetic algorithm and simulated annealing framework, as a larger set of independent moves is available during optimization. Other useful directions for further work include integrating the period graph model into the scheduling phase, rather than restricting its use to voltage scaling of fixed schedules, and the investigation of adaptive methods for dynamically adjusting the frequency of resimulation. The application graphs are shown in figures 8, 9, 10, 11, 12, 13, 14, and 15,.

## 10 References

[1] T. Back, U. Hammel, and H. Schwefel. "Evolutionary computation: Comments on the history and current state." *IEEE Transactions on Evolutionary Computation,* April, 1997.

[2] N. K. Bambha and S. S. Bhattacharyya. A joint power/performance optimization technique for multiprocessor systems using a period graph construct. In *Proceedings of the International Symposium on Systems Synthesis*, Madrid, Spain, September 2000. To appear.

[3] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, January 1994.

[4] Carter, Everett, Taygeta Scientific Inc. http://www.taygeta.com/annealing/simanneal.html

[5] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. Brodersen, "Optimizing power using transformations," *IEEE Trans. Computer-Aided Design,* vol. 14, no. 1, 1995.

[6] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. "Low-power CMOS digital design." *IEEE Journal of Solid-*

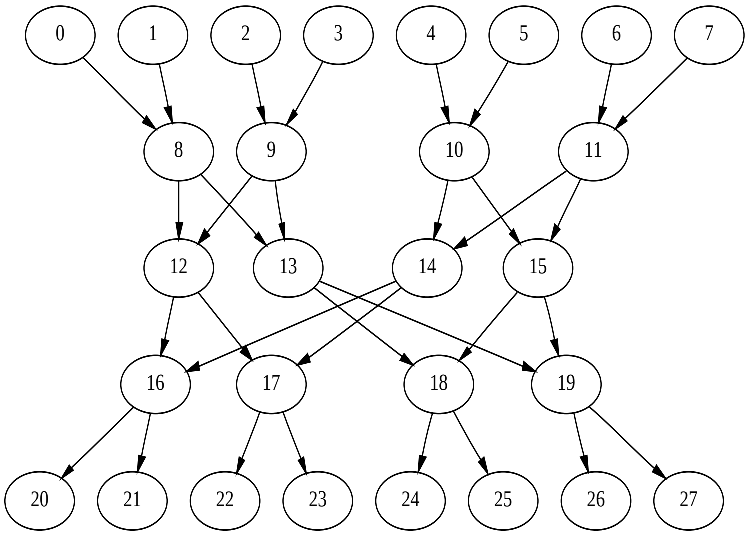


Figure 8. FFT2 application graph.

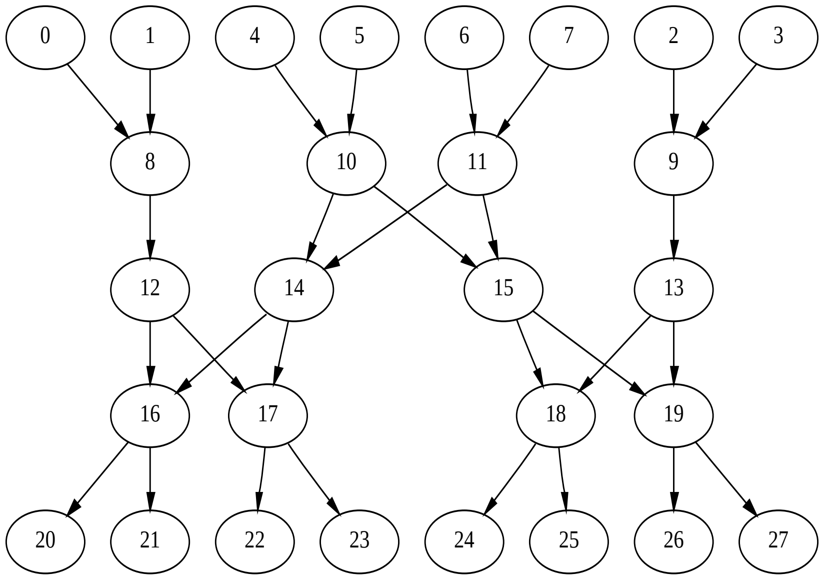


Figure 9.  FFT1 application graph.

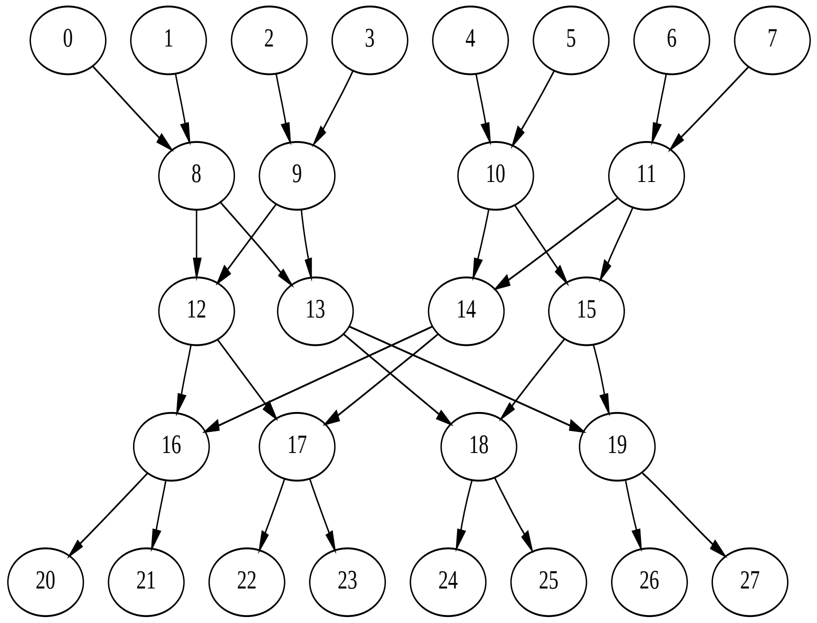


Figure 10. FFT3 application graph

.

| application | using period graph | no period graph |
|---|---|---|
| fft1 | 0.54 | 0.74 |
| fft2 | 0.69 | 0.86 |
| fft3 | 0.57 | 0.78 |
| mus | 0.68 | 0.90 |
| meas | 0.70 | 0.82 |
| qmf | 0.64 | 0.84 |
| rand1(30) | 0.55 | 0.78 |
| rand2(100) | 0.70 | 1 |
| rand3(200) | 0.87 | 1 |

Table 2. Genetic algorithm (optimized power)/(initial power) for fixed compile time

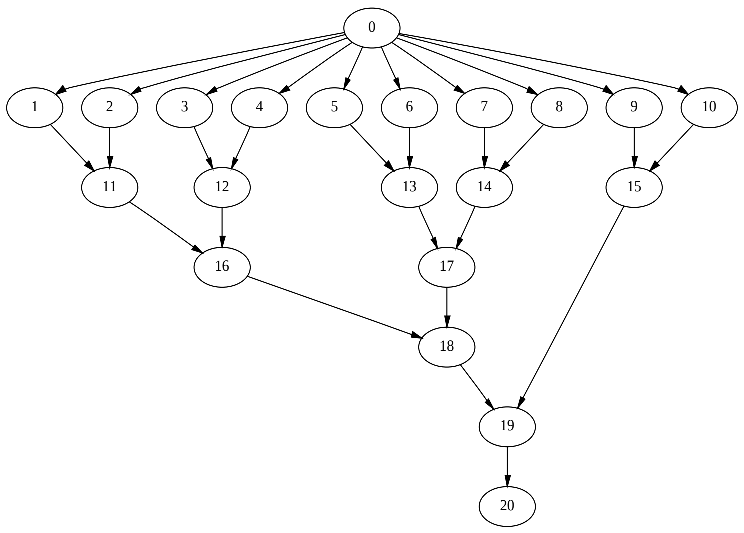


Figure 11. Karplus Strong music (mus) application graph.

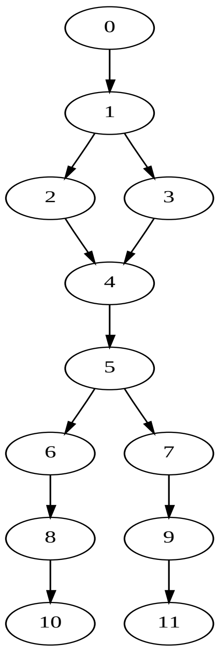


Figure 12. Meas application graph.

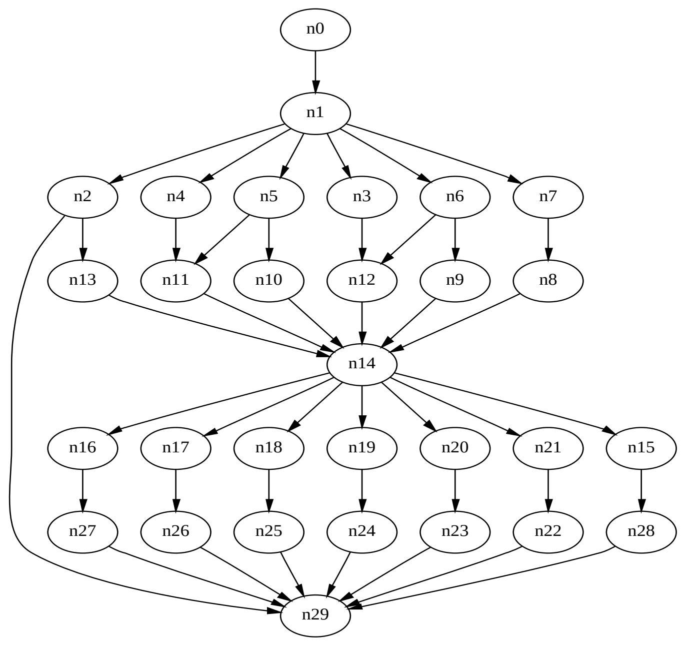
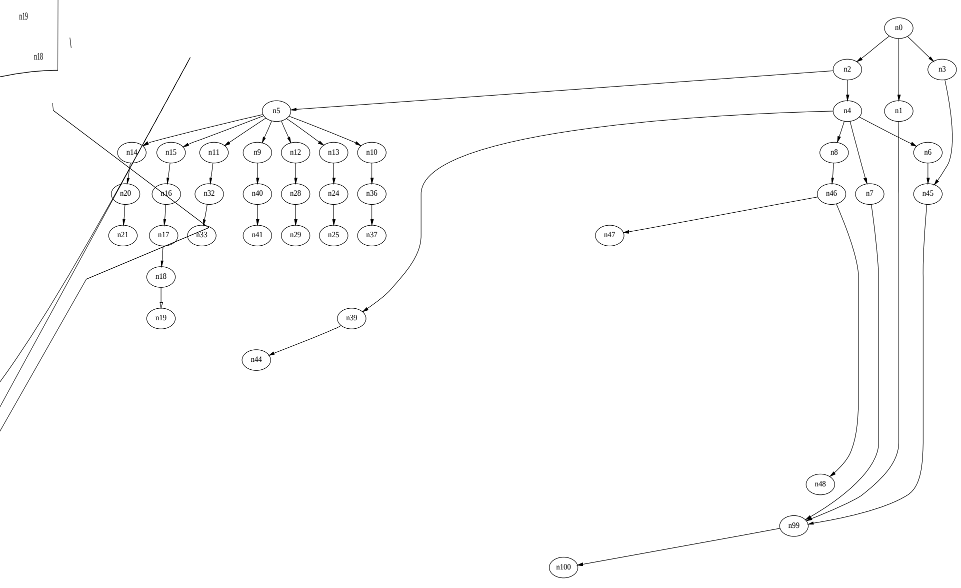


Figure 13. rand1 (30 nodes) application graph.



*State Circuits*, 27(4):473—484, 1992.

[7] J. M. Chang and M. Pedram, “Register allocation and binding for low power,” *Design Automation Conf.*, June, 1995.

[8] A. Dasgupta and R. Karri, “Simultaneous scheduling and binding for power minimization during microarchitecture synthesis,” in *Proc. Intl. Symp. Low Power Design,* Apr. 1995.

[9] L. Goodby, A. Orailoglu, and P. M. Chau, “Microarchitectural synthesis of performance-constrained low-power VLSI designs,” in *Proc. Int. Conf. Computer Design,* Oct. 1994.

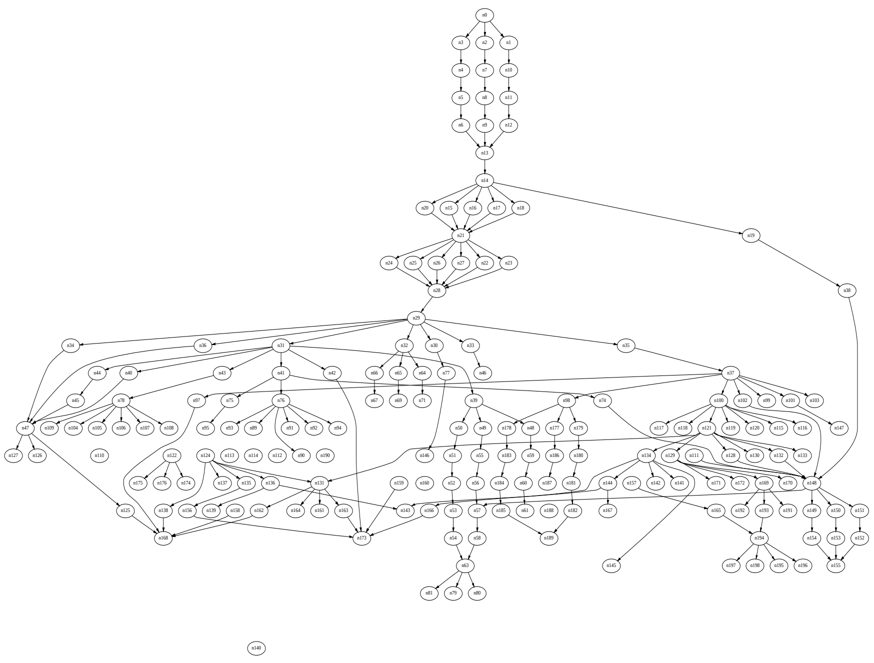
[10] C. Hwang and A.C.-H. Wu. “A predictive system shutdown method for energy saving of event-driven computation.” *International Conference on Computer-Aided Design*, 1997.

[11] M. Khandelia and S. S. Bhattacharyya. Contention-conscious transaction ordering in embedded multiprocessors. In *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors*, pages 276-285, Boston, Massachusetts, July 2000.

[12] E. L. Lawler. *Combinatorial Optimization*. Holt, Rinehart and Winston. 1976.

[13] E. A. Lee and S. Ha. Scheduling strategies for multiprocessor real time DSP. *Global Telecommunications Conference*, November 1989.

[14] Y. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the Design Automation Conference*, 1995.



[15] P. Macken, M. Degrauwe, M. Van Paemel, and G. Oguey, “A voltage reduction technique for digital systems,” in *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1990.

[16] A. Raghunathan and N. K. Jha, “Behavioral synthesis for low power,” in *Proc. Intl. Conf. Computer Design,* Oct. 1994.

[17] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):75-87, February 1993.

[18] G. C. Sih, “Multiprocessor Scheduling to Account for Interprocessor Communication”, Ph.D. thesis, Dept. of EECS, U. C. Berkeley, 1991.

[19] S. Sriram, and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000

[20] M. Srivastava, A. P. Chandrakasan, and R.W. Brodersen. “Predictive system shutdown and other architectural techniques for energy efficient programmable computation.” *IEEE Transactions on VLSI Systems*, 4(1): 42—55, 1996

[21] V. Tiwari, S. Malik, and A. Wolfe, “Power Analysis of Embedded Software: A First Step Towards Software Power Minimization”, *IEEE Trans. VLSI,* December 1994.