In Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation, Samos, Greece, July, 2002.

Goal-Driven Reconfiguration of Polymorphous Architectures

Sumit Lohani and Shuvra S. Bhattacharyya

Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland, College Park MD 20742, USA
{slohani, ssb}@eng.umd.edu

Abstract. Polymorphous computing architectures refer to computing platforms whose computation and communication structures can be changed over time. The objective of such platforms is to use the underlying reconfigurable components and attributes to adapt to dynamically changing constraints, objectives, and characteristics in the applications that execute on them. In this paper, we present a model for executing applications with non-deterministic execution times and time-varying performance requirements on a polymorphous architecture. We analyze the complexity of various issues related to the model, and identify some broadly-applicable conditions under which the complexity of these issues can be reduced. We develop a heuristic framework for guiding the run-time configuration adaptation process, and show through simulation experiments that this approach can efficiently handle both dynamics in performance requirements and in task execution times.

1. Introduction

There have been many advancements in recent years on architectures for reconfigurable processing engines (e.g, see [7][12][13]). With the increasing degree of reconfigurability in processing architectures, it is useful to view embedded multiprocessor systems as polymorphous computing architectures (PCAs) in which configurable attributes of the architecture and software are adapted in response to dynamically changing needs. Such attributes may include items such as inter-processor message routing, caching policies, scheduling policies, processor voltages, resource allocation to computing units, and synchronization protocols. A PCA can be a particularly useful platform for developing a computing system where applications and performance requirements change at run-time as one can adaptively configure the PCA to suit the dynamic constraints and objectives.

This paper takes a step towards bridging techniques for scheduling and system synthesis with reconfigurable processing platforms and the dynamically-changing application requirements that drive these platforms. We first formulate the problem of executing application dataflow graphs on a polymorphous computing architecture such that specified performance requirements are satisfied, where the requirements may vary over time and the application may have tasks with non-deterministic execution times (e.g., due to data dependencies or unpredictable events such as cache misses and interrupts). We analyze key properties of this problem and the complexity of some relevant sub-problems. We then develop a flexible heuristic framework for guiding the run-time configuration adaptation process, and show through simulation experiments that this approach can efficiently handle both dynamics in performance requirements and dynamics in task execution time behavior.

In the application model addressed in this paper, computational tasks (*actors*), which are represented by dataflow graph vertices, in the application are allowed to have stochastic execution times with static distributions or distributions that may vary slowly over time. The computing unit is a reconfigurable multiprocessor architecture, and the objective is to find a mapping of the actors in the application onto the processors in the multiprocessor and the configuration that the architecture should assume such that performance-related constraints (e.g., constraints on power, resource usage or throughput) are satisfied and objectives (e.g., maximizing throughput or minimizing latency) are optimized effectively. Furthermore, the constraints and objectives may vary over time, and thus, overall solution quality can be viewed in terms of how efficiently reconfiguration of the architecture tracks changes in the application's requirements. Henceforth, we will refer to this problem as the polymorphous computing architecture mapping (PCA mapping) problem. As can be seen, the PCA mapping problem is quite general in nature and even very restricted special cases can be proved to be NP-complete.

The approach suggested in this paper is correspondingly general and can handle diverse applications and performance requirements. All the reported experiments were performed on an abstraction of the Raw architecture [12] that incorporates salient features of the architecture such as the programmability of interconnects between processors. For experiments, the self-timed execution of applications on this abstracted Raw architecture was simulated using the inter-processor communication (IPC) graph model [12].

The emphasis in this paper is on coordination of the on line configuration management process for reconfigurable networks of processors, rather than the development of specialized configuration optimization techniques (such as fixed-objective scheduling and allocation), which are already in abundance in the literature (e.g., see [11] for a survey). Our work is complementary to such existing efforts and also to work on multiprocessor system synthesis [1][2], which can be used to derive the store of pre-computed configurations that is input to the techniques developed in this paper.

2. Problem formulation

A set of relevant metrics, such as latency, throughput, average power, peak power, and number of resources, is denoted by M. If a certain metric appears as a constraint with a value to be satisfied when the application executes, then this metric is referred to as a *constraint metric* and the value as a *constraint value* for that particular metric. A constraint value belongs to the set of real numbers. A pair of constraint metric and constraint value is called a *constrain pair.* A sequence of constraint pairs in turn is referred to as a *constraint vector,* and is denoted by

$$V = [(m_1, c_1), (m_2, c_2), \ldots, (m_K, c_K)], \tag{1}$$

where $m_1, m_2, \ldots, m_K$ represent any K metrics in M, and $c_1, c_2, \ldots, c_K$ represent the corresponding constraint values, for $K \in \{0, N\}$, where N is the number of all constraint pairs. This (possibly empty) sequence of constraint pairs in a constraint vector is prioritized such that (m_i, c_i) is a higher priority constraint pair than a constraint pair (m_j, c_j) if $i < j$, for $i, j \in \{1, 2, \ldots, K\}$ in a constraint vector $V = [(m_1, c_1), (m_2, c_2), \ldots, (m_K, c_K)]$. A metric m_R that is to be optimized after all constraints have been satisfied is called a *residual objective*. A *goal* g is an ordered pair (V, m_R), where V is a constraint vector and m_R is a residual objective. If there is no residual objective, then the goal is composed of only a constraint vector and can be represented by $(V, \perp)$. Here, the symbol $\perp$ represents the absence of a residual objective. Also, without loss of generality, the metrics are such that the associated optimization problems are to *minimize* the metric (i.e., a lower value of a metric is always better than a higher value). Metrics for which higher values are more desirable must thus be transformed into corresponding metrics for which lower values are better. For example, in iterative applications, the throughput (average rate of completion of application iterations) can be re-cast as the *average iteration period*, which is the reciprocal of the throughput.

Example 1: Consider a set of relevant metrics $M = \{L, P, T\}$, where L is the latency, P is the average power consumption, and T is the iteration period. Consider the goal $g = [(L, 50), (P, 100), (L, 40), (P, 70), T]$. In g, the constraint pair $(L, 50)$ has higher priority than the constraint pair $(P, 100)$, which in turn has higher priority than the constraint pair $(L, 40)$. The metric T is the residual objective.

This definition of reconfiguration goals as prioritized lists with optional residual objectives leads to a view of dynamic reconfiguration as a sequence of one-dimensional optimization problems. This simplification is useful because run-time adaptation techniques must be of relatively low complexity, and thus, one-dimensional optimization is a better match. Additionally, it allows us to leverage existing libraries of single-dimensional synthesis techniques, which are more abundant than multi-dimensional techniques. Third, it provides an intuitive and unambiguous format for designers to prioritize multidimensional application requirements. Note, however, that this formulation applies only to run-time reconfiguration, and multi-dimensional optimization techniques, such as SPEA-based methods [14], can be used off-line in arbitrary ways to compute caches of pre-computed configurations. Use of such caches will be discussed further in Section 3.2-5.

For example, in Example 1, we initially have an unconstrained latency optimization problem (since the first constraint involves latency). As we adapt the system configuration with techniques that address this problem, we will in general improve the latency. Once the latency improves to 50 time units, the current constraint is satisfied, and we switch to a power-optimization problem subject to a constraint of $L = 50$. The optimization process may continue in this manner until the last constraint is satisfied (in this case, $P = 70$), at which point run-time adaptation stops (if there is no residual objective) or reaches a terminal mode of optimizing the residual objective subject to all constraints in the constraint vector. This mode then continues until the system shuts down or the application's goal changes.

Mapping an application to a multiprocessor architecture includes defining a task-to-processor mapping along with defining the configuration of the reconfigurable architecture. In this paper, the scope of the word "configuration" is expanded to include also the mapping of the application onto the reconfigurable architecture.

Therefore, a *configuration* consists of two components 1) task-to-processor mapping and 2) configuration of the architecture. Henceforth, the word "configuration" is used in the above sense, unless stated otherwise. A given application, goal, and resource set define an *instance* of the PCA mapping problem. Input to the model is an instance that may change with time. We define the *design space* as the set of all feasible combinations of an instance and a configuration. The *solution space* for a feasible instance is the set of all feasible configurations for that instance. Latency, throughput, average power and peak power are some of the commonly encountered metrics. With many metrics of simultaneous relevance, the goal space is too vast to be fully explored before run-time, and run-time adaptation of configurations is generally advantageous.

Figure 1 illustrates a general model for solving the PCA mapping algorithm with a combination of off-line and on-line techniques. The main components of the model are the *off-line component,* the *configuration store* (*CS*), and the *on-line component*. The off-line component, whose objective is to pre-compute a set of efficient candidate mappings for various run-time scenarios, can be constructed using existing methods for scheduling, system synthesis, and multi-objective optimization. The focus of this paper is thus on the on-line refinement component and its interaction with the configuration store.

For a given instance, not every configuration is suitable as some configurations may violate constraints or may not adequately address residual objectives. As the goal changes for a given application, the system needs to derive a suitable adaptation of the run-time configuration. Optimally solving this problem is undecidable in many contexts. Also, reconfigurability of the architecture and the stochastic variance of execution times greatly complicates the solution space consisting of all possible configurations for the input of a goal and a given application. Since computing a suitable configuration is performed during the execution of an application, one can not apply exhaustive or relatively sophisticated search strategies as those techniques will take away excessive computational resources away from the application itself. To address this trade-off (thoroughness of dynamic optimization vs. resources drained from the application), our model of the PCA mapping problem also accounts for the time spent in computing efficient adaptations of mappings at run-time on the basis of feedback obtained from execution and identification of bottlenecks, and hence always tries to move towards an optimal solution. This is taken care of in the on-line refinement part of the model, which consists of low-complexity algorithms that find and refine configurations for a given instance. It also consists of feedback units shown by the "*Identify bottlenecks*" block in Figure 1 that takes feedback from the execution of the configurations and modifies the configurations so as to better suit the active goal. The *OnlineStats* unit in the on-line refinement part of the model stores short-term statistics that can be used by on-line algorithms.

A configuration store is used to store high-quality points in the design space that have been explored so that one can use them later as need be. The off-line refinement part of Figure 1 consists of high-complexity algorithms that yield better solutions. It is acceptable for them to be of high-complexity as they are used off-line, and do not compete for resources with the application. In Figure 1, the *STATS* unit stores statistics about the application (e.g., distributions of execution times for different actors, frequencies of occurrence of some particular regions of the goal space, etc.). Off-line algorithms use these statistics to explore the solution space for input instances.

As soon as the goal or application changes, an initial configuration is found using the on-line configuration management component in conjunction with the configuration store. On-line algorithms keep improving the configuration that is being executed, using the feedback from the execution. In the meantime, off-line algorithms may keep exploring areas of design space and merge the relevant information into the configuration store (for use in the selection of future initial configurations).

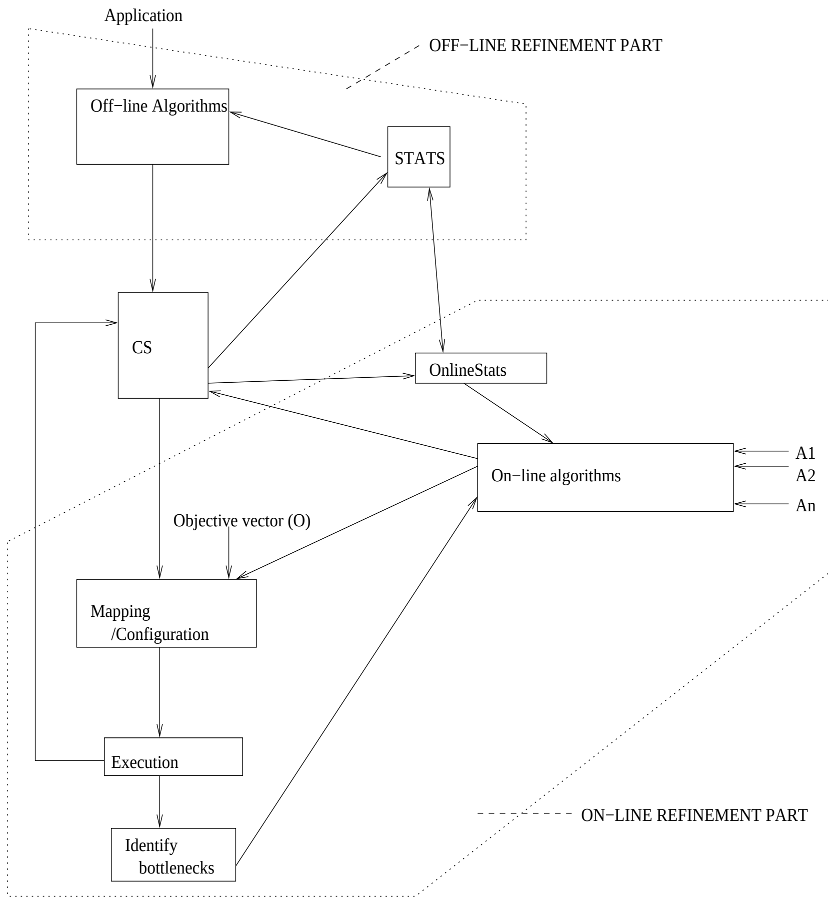

Fig. 1. An overview of the system-level reconfiguration framework studied in this paper.

3. Configuration management model

The overview of our PCA system synthesis model shows that it is very adaptive in nature and hence is suitable for applications with stochastic execution times and time-varying goals. This section develops further details of this model.

3.1 Evaluation of configurations and goals

It is useful to define some measure of how well a given configuration executes for a particular instance. This evaluation measure should allow unambiguous comparison between two configurations based on the current goal.

Suppose we are given a goal $g = [V, m_R]$, where

$$V = [(m_1, c_1), (m_2, c_2), \ldots, (m_n, c_n)] . \tag{2}$$

We define the *quality* of a system configuration C with respect to goal g, denoted $Q_g(C)$ (or simply $Q(C)$ if g is understood) as the ordered pair $Q(C) = (k, v)$, where $k + 1$ is the index of first unsatisfied constraint in the constraint vector of g (i.e., the lowest-index constraint in g that is not satisfied by the configuration), and v is the value obtained for the metric m_{k+1}. If configuration C satisfies all n constraints in the constraint vector of g, then we say that C *satisfies* g, and in this case, $Q(C) = (n + 1, v_R)$, where v_R is the value obtained for the residual objective m_R if $m_R \neq \perp$ or $v_R = -\infty$ if $m_R = \perp$.

In summary, the quality of a configuration measures a configuration with respect to a given goal, and given a goal and two configurations C_1 and C_2 with qualities $Q(C_1) = (k_1, v_1)$ and $Q(C_2) = (k_2, v_2)$ for that instance, respectively, C_1 has higher quality than C_2 if

$$(k_1 > k_2) \text{ or } ((k_1 = k_2) \text{ and } (v_1 < v_2)) . \tag{3}$$

3.2 Configuration store

A configuration store serves as a repository of alternative configurations. A configuration store can be divided into several sub-stores (sub-CSs), one for each relevant application. Each sub-CS has some configurations stored in it, one for a specific combination of goal and resource set. In the later part of this section, we assume that we are dealing with a fixed application and a fixed resource set, unless stated otherwise. This does not detract from the generality of the ideas developed later as they can be generalized to include various applications and resource sets using the hierarchical model of configuration store explained above.

Assuming a fixed application and resource set, selecting the goals whose corresponding configurations should be stored in the configuration store depends on various factors such as the size of the configuration store; the optimality of the stored configuration; computational resources drained from the application during execution by the on-line refinement algorithms; and the expected or observed frequency of specific goals.

3.3 Acceptability of configurations

Notions of *acceptability* and *cover* emerge naturally from this concept of configurations stores, and guide the construction and adaptation of the configuration store in our model. For example, one can envision the reconfiguration process as selecting an acceptable configuration, and gradually tightening the notion of acceptability to guide the on-line refinement process. The following definition makes these notions precise.

Definition 1: Given two goals g_1 and g_2, we say that g_1 is *acceptable* for g_2, denoted $g_1 \rightarrow g_2$, if a configuration that satisfies g_1 is an acceptable implementation for g_2. If $g_1 \rightarrow g_2$, we also say that g_1 *covers* g_2. Given a set Γ of goals and a specific goal g, the *space* of g over Γ (or simply, the *space* of g, if Γ is understood) is $\{g' \in \Gamma | g \rightarrow g'\}$. Thus, the space of a goal g is the set of goals that are acceptably implemented by any configuration that satisfies g. The space of a goal g is represented by $space(g)$.

The following result, proved and elaborated on in [9], shows that the acceptability of configurations is a particularly well-behaved relation if it is a partial order.

Theorem 1: If we have a finite set Γ of relevant goals, and the acceptability relation is a partial order, then there exists a unique, minimal set of goals $\{g_1, g_2, \ldots, g_n\}$ such that

$$\bigcup_{i=1}^{n} space(g_i) = \Gamma, \tag{4}$$

and this set of goals can be computed in polynomial time in $|\Gamma|$, the number of relevant goals.

Definition 2: *Dominance relation*: A point $p \varepsilon \Re^n$ dominates a point $q \varepsilon \Re^n$ if $p_i \le q_i$, for all $i = 1, \ldots, n$, where p_i and q_i denote ith components of p and q, respectively.

One can see that the dominance relation is a partial order [5]. We can have an acceptability relation between goals based on the dominance relation where a goal g_1 is acceptable for a goal g_2 if the constraint vector of the goal g_1 dominates the constraint vector of the goal g_2, and the residual objectives for both goals are same. The following example illustrates an acceptability relation that is not a partial order.

Example 2: Suppose that we have a single constraint metric, which is the average iteration period T of the system. Thus, the constraint associated with a goal g can be expressed as the desired average iteration period $T(g)$. Suppose that in a particular implementation context, the acceptability relation $g_1 \rightarrow g_2$ is defined by $T(g_1) - T(g_2) \le \Delta T$ for some positive real number ΔT. Thus, a configuration for g_1 can be worse than what is desired under g_2, and still be acceptable for g_2, as long as the deviation does not exceed the threshold ΔT. Suppose also that the goals g_1, g_2 and g_3 have desired average iteration period values of

$$T(g_1) = 5, \; T(g_2) = 5 - \frac{3\Delta T}{4} \text{ and } T(g_3) = 5 - \frac{3\Delta T}{2}. \tag{5}$$

One can then see that $g_1 \rightarrow g_2$ and $g_2 \rightarrow g_3$ but g_1 is not acceptable for g_3. Therefore, this acceptability relation is not transitive, and thus, is not a partial order.

An acceptability relation between goals based on the dominance relation or any other partial order leads to valuable properties such as that exposed by Theorem 1.

Also, the dominance relation is a natural candidate for an acceptability relation among goals, as a configuration corresponding to the dominating goal can be used in place of a configuration corresponding to the dominated goal without violating any constraints. This motivates our use of the dominance relation in managing configuration stores. One can observe that our approaches of defining a goal and the quality of a configuration are all consistent with acceptability based on the dominance relation.

4. On-line configuration management

In this section, we define an on-line configuration management framework called *CMF* that defines how to choose an initial configuration for a particular instance, and how the on-line adaptation for that configuration should proceed. We also formulate problems related to storage of configurations in the configuration store. These problems and our models to solve them provide fundamental analysis of the complexity of configuration management and provide feasible, low-complexity solutions to this problem.

A pseudocode outline of the CMF approach is shown in Figure 2. The objective is to provide a framework that imposes minimal constraints on how reconfiguration is actually performed, while providing systematic support for managing the reconfiguration process in terms of configuration stores, performance constraints, and optimization objectives. CMF is a meta-algorithm because specific details of the architecture, the application, and the on-line adaptation algorithms are left unspecified, and can be customized based on the relevant classes of applications and architectures. This meta-algorithm maintains a *current objective* at all times, where the goal is always to improve the current objective without violating any of the previously satisfied constraints. The function *onLineAdaptation* takes an objective metric, a constraint value, and a configuration as inputs, and keeps refining the configuration in an effort to continually improve its quality (as defined in Section 3.1). This function would typically be called within an enclosing loop that performs any system-dependent re-initialization and re-invokes the function immediately after the previous invocation of the function terminates (observe that the function terminates when the current goal is changed).

Pseudocode for the related functions is given in Figure 3. We have implemented CMF, and simulation results pertaining to it are discussed in Section 5.

4.1 Issues related to configuration management

Before proceeding with discussion of our experiments with CMF, we first study some fundamental versions of the problems related to configuration management, discuss their complexity, and relate aspects of them to well-studied problems. Two related problems regarding the size of the configuration store are as follows.

P1. Find the minimum size configuration store and the goals that should be stored in it such that all the relevant goals are covered.

P2. If one has a well-defined measure of “distance” between goals and the goal-pace is a metric space [4], then for a given fixed size configuration store, find the goals whose configurations should be stored such that the sum of the distances of those goals that are not present in the configuration store, from the distance-wise nearest goal present in the configuration store, is minimum.

```
function CMF

/* Global variables accessed: the current goal and the set
of pre-computed configurations, respectively.
*/

global goal $g_c = [(m_1, c_1), (m_2, c_2), \ldots, (m_K, c_K), m_R]$
global configurationStore C

stack $S$ = emptyStack ;
goal $g$, $g_o$;
$g = g_o = g_c$

/* The current optimization metric and the current
constraint to satisfy
*/
objective $objective_c = m_R$
constraint $constraint_c$ = null

$\sigma = \{c \in C | c \text{ satisfies } g\}$
while ($\sigma = \varnothing$) {
      ($g$, $constraint_c$, $objective_c$) = demoteConstraint($g$, $S$)
      $\sigma = \{c \in C | c \text{ satisfies } g\}$
}

/* Select an admissible configuration from σ using some
heuristic or specialized, optimal algorithm.
*/
$configuration_c$ = select(σ)

/* Keep trying to refine the current configuration accord-
ing to the current goal until the goal changes ($g_c$ may
change at any time under external control).
*/
while ($g_c = g_o$) {
      while ($constraint_c$ is not satisfied by $configuration_c$) {
            onLineAdaptation($objective_c$, $constraint_c$,
                        $configuration_c$)
      }
      /* Move to the next unsatisfied constraint or to
      the residual objective */
      ($g$, $constraint_c$, $objective_c$) = promoteConstraint($g$, $S$)
}
end function
```

Fig. 2. The CMF framework for goal-driven reconfiguration.

P1 and P2 can be viewed, respectively, in terms of the well-known problems of minimum dominating sets and k-medians. To reduce P1 from the minimum dominating set problem [5], for every vertex in the dominating set problem, instantiate a goal, and for every edge, instantiate a condition that the goal corresponding to the source vertex is acceptable to the goal corresponding to the sink vertex. The problem P1 related to this set of goals and the acceptability relation among goals is equivalent to the given minimum dominating set problem instance. The vertices in the given minimum dominating set problem instance, corresponding to the goals that should be stored in the configuration store (found by solving P1) constitute a minimum dominating set for the given minimum dominating set problem instance. This can be used to show that the problem P1 is NP-hard (see [9] for more details). However, if the acceptability relation is a partial order, then the minimum dominating set can be found in polynomial-time by picking up all the vertices with no incoming edges in the graph of the minimum dominating set problem. This is in accordance with Theorem 1, and further underscores the advantage of using acceptability relations that are partial orders.

If the associated distance function is defined between any two goals and the goal-space is a metric space, then problem P2 can be modeled in terms of the *k-median*

```
function promoteConstraint
input goal $g = [(m_1, c_1), (m_2, c_2), \ldots, (m_{K-1}, c_{K-1}), m_K]$, stack $S$
output goal, constraint, objective
goal $g'$
constraint $v = S.\mathrm{pop}()$
objective $m = S.\mathrm{pop}()$
constraint $x = S.\mathrm{pop}()$
$S.\mathrm{push}(x)$
$g' = [(m_1, c_1), (m_2, c_2), \ldots, (m_{K-1}, c_{K-1}), (m_K, v), m]$
return {$g'$, $x$, $m$)
end function
```

```
function demoteConstraint
input goal $g = [(m_1, c_1), (m_2, c_2), \ldots, (m_K, c_K), m_R]$, stack $S$
output goal, constraint, objective
goal $g'$
$S.\mathrm{push}(m_R)$
$S.\mathrm{push}(c_K)$
$g' = [(m_1, c_1), (m_2, c_2), \ldots, (m_{K-1}, c_{K-1}), m_K]$ .
return ($g'$, $c_K$, $m_K$)
end function
```

Fig. 3. Definition of functions *promoteConstraint* and *demoteConstraint* from Figure 2.

problem [2, 8], as shown in [9]. For the simple case of a two-dimensional goal space, a polynomial-time approximation algorithm with a 3-approximation factor exists for the k-median problem [2].

Configuration management problems P1 and P2 can be viewed as extreme cases in the sense that in one of them we want to cover all feasible goals without considering how large the minimum size configuration store would be (P1), and in the other case, we have a fixed size configuration store and we are trying to find out the maximum number of goals that can be covered using that configuration store even though that number could be much less than the total number of relevant goals (P2). A more elaborate formulation would be one in which we have to pay extra cost for increasing the size of the configuration store, but we would be gaining some additional service by that by being able to store more goals in the configuration store. This way we can explore various trade-offs between the size of the configuration store vs. the number of goals stored in a well-defined way. For the specific case when a distance function is defined between any two goals and the goal-space is a metric space, these trade-offs can be explored by modeling this problem as a facility-location problem [4, 8, 10], as explained in [9]. A polynomial-time algorithm with an approximation guarantee of 1.74 exists for the facility location problem [4].

5. On-line adaptation

In this section, we focus on the metrics of throughput and power consumption, and develop low-complexity, on-line strategies based on heuristics for throughput optimization and power optimization as implementations of the function *onLineAdaptation* in Figure 2. The objective is to demonstrate the efficacy of the CMF model, and show that it can produce efficient tracking of time-varying application requirements.

The approach of taking feedback from the execution of the application makes these on-line methods able to handle even applications with stochastic execution times that have time-varying distributions, in addition to applications with fixed execution times, and applications with stochastic attributes that have stationary distributions. In general, this on-line refinement formulation can thus be viewed as an approach to tracking the dynamics of the goal and the characteristics of the application.

To experiment with CMF, we used a simple heuristic based on load balancing [15] to optimize throughput during online adaptation. Pseudocode for this heuristic is represented by function *adaptThroughput* in Figure 4. In the pseudocode, $moveTask(c, n)$ is a function that chooses n tasks from a maximally loaded processor in a configuration c, and randomly, moves them to appropriate locations on a minimally loaded processor, and returns the modified configuration. Randomization in choosing tasks from the maximally loaded processor provides a low-complexity approach to increase the explored region of the design space and to calibrate the configuration to dynamic application characteristics. The function $executeTr(c, l)$ is a function that executes the application according to configuration c for a time interval of length l, and returns the throughput of the application during that interval. The value of l to use depends on the non-determinacy of the application. We define the non-determinacy of an application in the following way.

Let the number of possible execution times taken by an actor i be denoted by n_i.

We denote the set of n_i possible execution times taken by an actor i as $\{t_{i1}, t_{i2}, \ldots, t_{in_i}\}$. The probability of occurrence of a possible execution time t_{ik} for actor i, is denoted by p_{ik}, for all $k = 1, \ldots, n_i$. The *degree of non-determinacy* λ is a measure of the overall amount of non-determinacy in the application, specifically, in the actor execution times, and is defined as

$$\lambda = \frac{\sum_i \left\{ \sum_{k=1}^{n_i} \left\{ p_{ik}(t_{ik} - t_{i,mean})^2 \right\} \right\}}{\sum_i \{t_{i,mean}\}^2}, \tag{6}$$

where $t_{i,mean}$ denotes the mean execution time of actor i, and is defined as

```
/* This function adapts the given input configuration
      while executing the application.
*/
function adaptThroughput
input configuration $c$
global constant time $timelimit$, time $l$

time $t_{old} = \text{executeTr}(c, l)$
time $t$
configuration $c_{old} = c$
$n = 1$
while (clock < $timelimit$) {
        $c = \text{moveTask}(c_{old}, n)$
        if (exhausted all $n$-task movements
                    without improvement){
                $n = n + 1$
                $c = \text{moveTaskTr}(c_{old}, n)$
        }
        $t = \text{executeTr}(c, l)$
        if($t \geq t_{old}$) {
                $c_{old} = c$
                $t_{old} = t$
                $n = 1$
        }
        clock = clock + $l$
}
end function
```

Fig. 4. An online adaptation approach for throughput optimization.

$$t_{i,mean} = \left(\sum_{k=1}^{n_i} t_{ik} \right) / n_i . \qquad (7)$$

Generally, the more non-deterministic the application is, the longer it needs to be executed to determine an accurate value of average throughput.

The function *adaptThroughput* returns a configuration that it deems most appropriate for throughput maximization. Note that if moving any single task from the maximally loaded processor to the minimally loaded processor does not improve performance then the heuristic chooses a *pair* of tasks to be moved to another processor. This approach of progressively increasing the number of tasks to be moved continues whenever all combinations for a particular number of tasks have been exhausted. This approach thus attempts to make small low-complexity changes first and if that does not improve performance, the approach gradually reaches towards higher-complexity changes. The higher complexity changes are larger in number than small, low-complexity changes, and help the system in escaping from local minima.

In our experiments, inter-processor communication (IPC) per time unit during the execution is taken as an estimate for relative power consumption. Since IPC consumes relatively large amounts of power, it is a reasonable approximation for comparing the power consumption levels of alternative configurations on a homogeneous multiprocessor. To find a configuration that reduces the power consumption, we use an approach (called *adaptPower*) similar to the *adaptThroughput* approach used for throughput optimization, except that the probability of a task on a maximally loaded processor being transferred to a minimally loaded processor depends upon the IPC associated with that task. The higher the IPC associated with a task, the higher its chances are of being transferred to another processor.

6. Experimental results

An on-line adaptation scheme for refining a given goal is specified in Figure 5, and it is represented as function *onlineAdaptation* in the CMF pseudocode of Figure 2. In Figure 5, the appropriate online optimization strategy, such as the *adaptThroughput* or *adaptPower* approaches discussed above, is selected depending on the current optimization objective and system state. Typically, this strategy will be drawn dynamically from a library of simple, low-complexity techniques.

Table 1 shows the performance of our implementation of CMF using the heuristics developed in Section 5 for throughput optimization and power optimization based on various goals applied to several DSP benchmarks, including fast Fourier transform, filter bank, music synthesis, and measurement applications. The starting configuration that is refined is found by using standard critical path scheduling. The critical path length is computed in terms of average execution times of actors. The set of relevant metrics M for our experiments is $M = \{T, P\}$, where T denotes the average iteration period of the execution and P denotes the average power consumption. Experiments are reported for the following eight goals.

g_1 = {(P, 0.270), (T, 265), (P, 0.250), (T, 0.255), P}

g_2 = {(T, 260), (P, 0.240), T}
g_3 = {P, 0.125), (T, 180), P}
g_4 = {(T, 165), (P, 0.110), (T, 160), P}
g_5 = {(T, 360), (P, 0.160), (T, 355), (P, 0.155), (T, 350), P}
g_6 = {(T, 345), P}
g_7 = {(T, 215), (P, 0.040),T}
g_8 = {(P, 0.053), (T, 215), (P, 0.050), (T, 210), P}

In Table 1, the column titled "Goal" represents the goal that is applied to the application. Also, for a non-negative integer k, column v_k denotes the value of a metric of the best configuration found by the on-line adaptation scheme, after k configurations have been assessed by executing them for some time. For the same experiments that are reported in Table 1, Table 2 shows the times at which different constraints associated with the applied goals are satisfied. For a given goal that is applied to an application, n_i denotes the number of configurations that have been executed in order to assess them before the ith constraint in the applied goal is satisfied. One can see

Application	λ	Goal	Metric	v_0	v_{10}	v_{20}	v_{30}	v_{40}	v_{50}	v_{60}
fft1	0	g_1	T	278	278	278	278	256	254	254
			P	.273	.269	.269	.269	.204	.226	.226
fft1	.359	g_2	T	309	256	251	251	251	252	259
			P	.242	.282	.278	.278	.278	.257	.221
qmf	0	g_3	T	145	242	198	198	186	170	170
			P	.133	.117	.098	.098	.088	.096	.096
qmf	.256	g_4	T	142	164	162	162	153	153	153
			P	.136	.127	.110	.110	.110	.110	.110
karp	0	g_5	T	395	353	346	342	342	342	342
			P	.131	.158	.156	148	.148	.148	.148
karp	.309	g_6	T	450	352	300	342	342	346	346
			P	.115	.155	.159	.151	.151	.148	.148
meas	0	g_7	T	220	212	201	184	184	184	184
			P	.054	.075	.059	.021	.021	.021	.021
meas	.405	g_8	T	185	218	212	212	212	210	196
			P	.064	.018	.037	.037	.037	.019	.040

Table 1. Experimental results for CMF.

```
function onLineAdaptation
input $objective_c$ , $constraint_c$ , $configuration_c$
global goal $g$ , $g_c$ , $g_o$
global constant time $timelimit$
global stack $S$ /* constraint stack */

time $t = 0$
while ($t < timelimit$ and $g_c = g_o$){
      while $constraint_c$ is not satisfied {
            if ($objective_c = throughput$) {
                  adaptThroughput($configuration_c$)
            } else if ( $objective_c = power$ ) {
                  adaptPower($configuration_c$)
            } else if …
                  … /* adapt for other objectives */ …
            }
      }
      ($g$, $constraint_c$, $objective_c$) = promoteConstraint($g$, $S$)
}
end function
```

Fig. 5. On-line adaptation scheme. This is an elaboration of function *onLineAdaptation*, which is called in Figure 2. It is effectively a wrapper for specialized reconfiguration optimizations.

App.	λ	Goal	n_1	n_2	n_3	n_4	n_5
fft1	0	g_1	1	37	39	42	-
fft1	.359	g_2	7	56	-	-	-
qmf	0	g_3	8	48	-	-	-
qmf	.256	g_4	0	13	36	-	-
karp	0	g_5	4	7	9	28	28
karp	.309	g_6	16	-	-	-	-
meas	0	g_7	8	28	-	-	-
meas	.405	g_8	3	17	17	48	-

Table 2. Results for CMF tracking an applied goal.

that in these experiments, CMF is able to meet the constraints specified in the given goals within a reasonable number of configurations.

7. Conclusion

In this paper, we have developed a framework called CMF for on-line adaptation of system-wide configurations of embedded multiprocessors. The objective is to provide a framework that imposes minimal constraints on how reconfiguration is actually performed (i.e., the specific optimization algorithms that are used during off-line and online configuration synthesis), while providing systematic support for managing the reconfiguration process in terms of configuration stores, performance constraints, and optimization objectives. The CMF approach is shown to be effective through analysis and experimental results on several DSP benchmarks, which demonstrate the ability of CMF to systematically adapt system configurations towards progressively better solutions for a variety of goals, even in the presence of significant uncertainties in task execution times.

8. References

1. S. S. Bhattacharyya. *Hardware/software co-synthesis of DSP systems.* In Y. H. Hu, editor, *Programmable Digital Signal Processors: Architecture, Programming, and Applications*, pages 333-378. Marcel Dekker, Inc., 2002
2. T. Blickle, J. Teich, and L. Thiele. System-level synthesis using evolutionary algorithms. *Journal of Design Automation for Embedded Systems*, 3(1):23-58, 1998.
3. M. Charikar and S. Guha. improved combinatorial algorithms for facility location and k-median problems. *Proc. 40th Annual Symposium on Foundations of Computer Science*, 378-388, 1999.
4. F. Chudak, "Improved approximation algorithms for uncapaciateted facility location", In R. E. Bixby, E. A. Boyd and R. Z. Rios-Mercado, eds., *Integer Programming and Combinatorial Optimization*, Springer LNCS Vol. 1412, 180-194, 1998.
5. T. Cormen et al. *Introduction to Algorithms*, McGraw Hill, 2000.
6. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman and company, 1999.
7. J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 24-33, April 1997.
8. K. Jain and V. V. Vazirani, "Approximation algorithms for metric Facility location and k-median problems using the primal-dual scheme and Lagrangian relaxation", *Proc. Foundations of Computer Science, 1999.*
9. S. Lohani and S. S. Bhattacharyya. System synthesis for polymorphous computing architectures. Technical Report UMIACS-TR-2002-12, Institute for Advanced Computer Studies, University of Maryland at College Park, February 2002. Also Computer Science Technical Report CS-TR-4330.
10. D. B. Shmoys, E. Tardos, and K. I. Aardal. Approximation algorithms for facility location problems. *Proc. 29th ACM Symp. on Theory of Computing,* 265-274, 1997.
11. S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors:Scheduling and*

Synchronization, Marcel Dekker, 2000.

12. E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe and A. Agarwal, "Baring it all to Software: Raw Machines", *IEEE Computer,* September 1997, pp. 86-93.
13. S. Wong, S. Vassiliadis, and S. Cotofana. Microcoded reconfigurable embedded processors: Current developments. In *Proceedings of the International Workshop on System Architecture Modeling and Simulation*, pages 207-223, July 2001.
14. E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: A comparative case study and the strength Pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257-271, November 1999.
15. A. Y. Zomaya, "Parallel and Distributed Computing: The Scene, the Props, the Players," *Parallel and Distributed Computing Handbook,* A.Y. Zomaya, ed., pp. 5-23, New York: McGraw-Hill, 1996.