Direct Backtracking: An Advanced Adaptation Algorithm for Pervasive Applications

Stephan Schuhmann, Klaus Herrmann, Kurt Rothermel

University of Stuttgart
Institute of Parallel and Distributed Systems (IPVS)
Universitätsstr. 38
70569 Stuttgart, Germany
Email: {firstname.lastname}@ipvs.uni-stuttgart.de

Abstract. The adaptation of pervasive applications is in the focus of many current research projects. While decentralized adaptation is mandatory in infrastructureless ad hoc scenarios, most realistic pervasive application scenarios are situated in heterogeneous environments where additional computation power of resource-rich devices can be exploited. Therefore, we propose a hybrid approach to application configuration that applies centralized as well as decentralized configuration as appropriate in the given environment. In this paper we introduce the Direct Backtracking algorithm that represents an efficient way for centralized configuration and adaptation of pervasive applications in heterogeneous scenarios. In our evaluation, we show that compared with other centralized algorithms, our algorithm significantly reduces adaptation latency as it avoids unnecessary adaptations that arise in many other backtracking algorithms, without significantly increasing memory waste. This is achieved by introducing two mechanisms: 1. proactive backtracking avoidance and 2. intelligent backtracking.

1 Introduction

In recent years, automatic adaptation of pervasive applications that share the resources of different devices has become a research field of increasing interest. Besides our component system PCOM [1], many other research projects such as Gaia [2], Aura [3], or Pebbles [4] deal with this issue. Two fundamentally different approaches exist for adaptation of pervasive applications, namely distributed and centralized adaptation. The distributed approach is generally applicable in scenarios both with and without additional infrastructure. Unfortunately, this approach needs extensive communication between devices. Furthermore, it assumes a homogeneous environment and, thus, does not exploit additional computation power available on resource-rich devices.

Our work focuses on heterogeneous pervasive computing environments where resource-poor, potentially mobile devices as well as resource-rich infrastructure devices are present. Many of today's office or home scenarios satisfy these properties. In such scenarios, centralized adaptation can improve calculation speed

Published in U. Brinkschulte et al. (Eds.): ARCS 2008, LNCS 4934, pp. 53-67, 2008.

and decrease communication overhead dramatically. Due to the wide range of different scenarios, we advocate a hybrid configuration approach that makes use of decentralized as well as centralized algorithms to exploit the resources present in the environment as effectively as possible.

In this paper, we present a new centralized configuration algorithm called *Direct Backtracking* (DBT) that can be employed on resource-rich devices to configure pervasive applications on near-by resource-poor devices. This complements our previous work on purely decentralized configuration [5] and enables a range of configuration solutions with different degrees of decentralization since both approaches may be combined to produce a hybrid configuration system.

We present two new innovative mechanisms in backtracking: 1. *proactive backtracking avoidance* and 2. *intelligent backtracking*. Both are used in our *Direct Backtracking* algorithm to reduce configuration latency and avoid thrashing effects (repetitive unnecessary reconfigurations) dramatically. In our evaluation, we show that *Direct Backtracking* displays vastly improved performance compared to its nearest competitor (Synchronous Backtracking) and, therefore, enables fast configurations and adaptations in much larger pervasive applications. We also show that increased memory consumption of our algorithm does not limit its general applicability in heterogeneous environments.

The paper is structured as follows: After a presentation of our system model in Section 2, we take a look on related work in Section 3. Then, we explain our algorithm in detail in Section 4. This is followed by an evaluation in Section 5. Finally, we recapitulate our work in Section 6 and give a short outlook on future works.

2 System Model

In many pervasive computing scenarios, a single device cannot provide the entire functionality required by an application because of limited resources. Thus, the main characteristic of a *pervasive application* is the fact that the required functionality is distributed among multiple devices which have to collaborate.

We assume a component-based software model, i.e. an application consists of *components* and each component *instance* requires a certain amount of resources. Figure 1 shows our application model. An application is represented by a tree of interdependent components that is constructed by recursively starting the components required by a root instance, the so-called application *anchor*.

A single component is resident on a specific device which is represented by a *container* that carries a unique identifier (ID). The number of components per container is not restricted. Interdependencies between components as well as resource requirements are described by directed *contracts* which specify the functionality required by the parent component and provided by the child component of this contract. A parent component may have an arbitrary number of dependencies. Our algorithm follows a depth-first search approach. This means that the algorithm proceeds from the top to the bottom of the tree and, within a sublevel of the tree, from left to right. If there exists more than just one

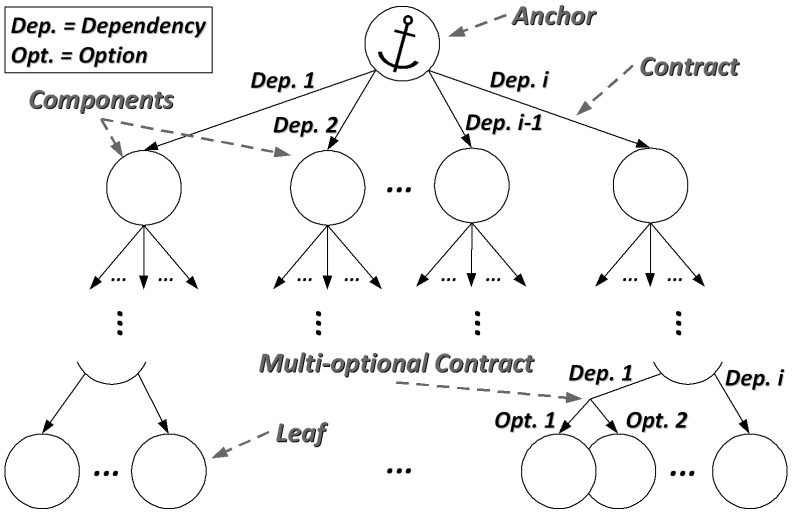

Fig. 1. Application model

component that provides the required functionality for a contract, the parent component can choose among several *options*. In the following, this is called *multi-optional contract*.

In a *configuration* process, the configuration algorithm tries to resolve all dependencies by finding a suitable component for each contract. Whenever the algorithm fails to find such a component for a contract (e.g., due to lack of resources), an *adaptation* process has to be initiated to resolve this conflict. Within this process, an instantiated component of another contract C has to be stopped to free resources, and another component that fulfills C's requirements with less resources has to be instantiated afterwards. If the algorithm does not consider whether the adapted contract requires the same resources as the contract which could not be fulfilled, it is possible that many adaptations are needless since they do not solve the conflict. So, the number of necessary adaptations increases which leads to additional configuration latency (time before the configuration is complete). This undesired effect is called *thrashing*.

As long as the anchor component is executed, its container ensures that dependencies are recursively resolved by binding adequate components to them. Since a parent component relies on its child components, it can only be instantiated if all of its children have been instantiated previously. An application is successfully started if all dependencies have been resolved so that for each contract, a suitable component which satisfies all requirements could be found.

We assume that a single device which performs centralized configuration and adaptation has collected relevant data beforehand using a specific protocol[1].

[1] This protocol is not further described here since it does not affect adaptation aspects and does not produce considerable overhead.

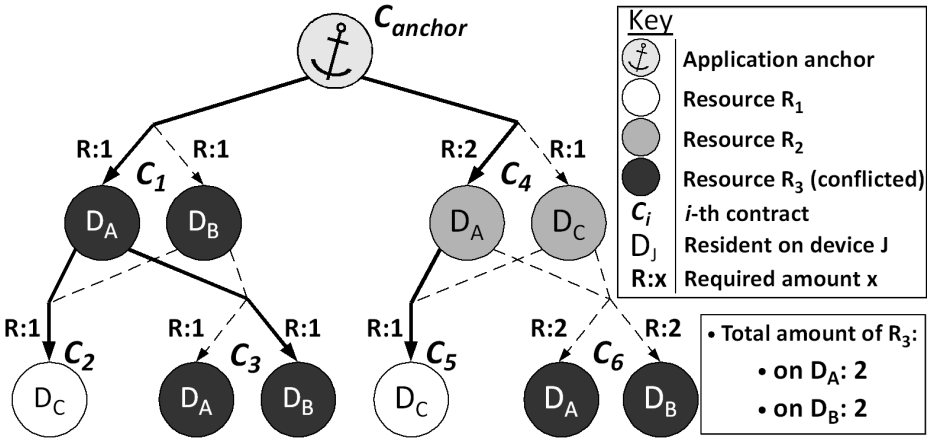

Fig. 2. Exemplary application tree

Thus, this device can create an internal representation of the application tree that comprises dependencies, components, and incorporated devices with their available resources. After a configuration is complete, another protocol has to distribute the results of the configuration to the respective devices.

2.1 Example

Figure 2 shows an exemplary application tree that consists of the anchor C_{anchor} and six contracts C_1 to C_6 whose dependencies have to be satisfied in order to start the application successfully. The contracts are ordered depth-first search like, according to the sequence of the algorithm. Hence, the algorithm starts the configuration process with C_1, and finishes with C_6. Contracts C_1, C_3, C_4, and C_6 are multi-optional, i.e. there exists more than one component that can satisfy the required functionality. The components are resident on three containers D_A, D_B, and D_C. The contracts can be satisfied by certain amounts of specific resources. For example, contracts C_1, C_3, and C_6 can be satisfied by R_3 which is available on containers D_A and D_B. Let us assume that enough resources of R_1 and R_2 are available on the containers and, thus, only R_3 can cause a conflict situation due to missing resources. Both D_A and D_B have two instances of R_3.

The thick lines show the components – resident on the denoted containers D_j – which were chosen by the algorithm for C_1 to C_5 and, thus, only C_6 has to be instantiated. In this situation, D_A as well as D_B have only one free resource of type R_3 left, but two resources are needed for every container to instantiate C_6. Therefore, none of the components can be instantiated for C_6 in this situation. Hence, an adaptation of the current configuration is inevitable.

As a concrete example, consider an instant messaging application that requires an input service, e.g. a keyboard or a touch screen, to write messages, and an output service to display messages. If there exists more than one input

device, for instance, these devices represent several options of a multi-optional contract. An adaptation is necessary, for instance, if the current output device becomes unavailable. Another example is a pervasive presenter [6] whose functionality of displaying presentation slides is provided by the cooperation of distributed devices.

The example described above could illustrate a similar presentation application, where resource R_3 represents display devices. D_A and D_B both have two displays. Displays are needed for different tasks within the application, e.g. to show images, videos, or presentation slides and, thus, are required in multiple contracts. Additionally required input and output devices like microphones or loudspeakers are represented by the other resources R_1 and R_2. Due to different properties of the resources, e.g. the maximum output volume of a loudspeaker, it is possible that different amounts of specific resources are required within a contract, as it can be seen for contract C_4.

3 Related Work

The adaptation of tree-based applications can be mapped to a Constraint Satisfaction Problem (CSP) [5]. In case of a fully distributed application configuration, this problem can be solved by distributed algorithms from the domain of Artificial Intelligence [7]. Due to their decentralization, these algorithms cause huge communication overhead for resolving dependencies between components. In case of n devices that are involved in the application configuration, the worst-case amount of messages to be sent is $\mathrm{O}(n^2)$. In centralized algorithms, every involved device has to send only one message to the configuration device to inform this device about its available resources, which leads to a worst-case message amount of only $\mathrm{O}(n)$. Furthermore, distributed algorithms do not take special care of resource-rich devices that are present in heterogeneous environments.

Thus, we concentrate on centralized backtracking algorithms in this paper. A survey of these algorithms is given by Baker [8]. The simplest centralized backtracking algorithm is *Synchronous Backtracking* (SBT) [9]. SBT executes a depth-first search in the application tree and has one huge drawback concerning adaptations: Since it does not consider the cause of a backtracking process and tries to adapt the first possible multi-optional contract, it suffers from thrashing. Especially for huge applications with many multi-optional contracts, this leads to an enormous overhead and, thus, increased latency.

Several approaches exist to avoid this thrashing effect. *Synchronous Backjumping* (SBJ) [10] searches for a multi-optional contract if it depends on the same kind of resource as the contract that was the reason for backtracking. It does not adapt contracts that are independent of the backtracking cause. This helps to reduce thrashing, but it cannot avoid it completely since SBJ does *not* keep previous intermediate results for subsequent adaptations. Furthermore, works with a stack during backtracking. This causes additional computation overhead.

Dependency-Directed Backtracking (DDB) [11] solves the problem of thrashing by storing a set of so-called *nogoods* which are partial configurations without a solution for the complete application. Therefore, it can avoid infeasible solutions subsequently. The drawback of DDB is its enormous memory consumption since an ever-increasing set of nogoods has to be stored on a stack.

An advanced approach to remove thrashing completely without excessive waste of memory is *Dynamic Backtracking* (DyBT) by Ginsberg [12]. Similar to SBJ and DDB, this algorithm immediately moves to a point which conflicts with the latest assignment in case of a conflict. DyBT neither forgets intermediate values, nor is it memory-intensive since it does not rely on a stack. It is an iterative algorithm that stores a set of so-called *culprits* which represent forbidden assignments. DyBT does not only retain the chosen value for a contract C, but also the culprit set of C. If the instantiation of a component for a contract fails, the algorithm can easily decide which of the formerly configured contracts conflict with this contract and directly jump back to them.

However, DyBT changes the order of contracts to resolve conflicts. This is not an option in our problem of tree-based application configuration since parent-child relationships and differences in resource consumption are encoded in the order of contracts. This order needs to be preserved to ensure useful configurations. So, DyBT cannot be used for configuration of tree-based pervasive applications.

Direct Backtracking proceeds similar to DyBT in general, but in addition, it also adapts the adherent subtree of a component during an adaptation process. Thus, Direct Backtracking does not need to perform any changes in the order of components. Moreover, our algorithm applies two intelligent mechanisms that render the configuration process more efficiently.

4 The Direct Backtracking Algorithm

In Section 4.1, the general configuration process of Direct Backtracking is described. This is followed by the main contribution of this paper which are two additional mechanisms of Direct Backtracking: a proactive mechanism to avoid backtracking that is described in Section 4.2, and an intelligent backtracking mechanism to handle conflict situations without thrashing which is presented in Section 4.3. Finally, Section 4.4 presents an example for the adaptation process.

4.1 General Approach

Figure 3 illustrates DBT in C-like pseudo code. DBT includes five different functions: *start*, *create*, *started*, *stopped*, and *backtrack*.

The *start* function initiates the configuration process and calls the *create* function to resolve the first dependency of the anchor. Thereby, the function `getChild(i, j)` returns the j-th option of the i-th dependency of the current component. The selection of an option in case of a multi-optional contract is performed within the function `determOption()`. This is carefully made to decrease

```
function start(Component anchor):
      create(getChild(0, determOption())); // see Sect. 4.2 for determOption()
end function;  // start

function create(Component comp):
      if (enoughResourcesAvailableForInstantiation())
            if (leafReached()) // leaf component can be started
                  started(comp.getParent());
            else  // try to instantiate first child component
                  create(comp.getChildComp(0, determOption()));
            end if;
      else  // component could not be instantiated
           stopped(comp.getParent());
      end if;
end function;  // create

function started(Component comp):
      if (unresolvedDependencyLeft())  // try to instantiate next dependency
            create(comp.getChild(comp.getDependency() + 1, determOption());
      else  // all dependencies resolved
            if (anchorReached())
                  terminateSuccessfully();  // configuration complete
            else  // indicate instantiation to parent component
                  started(comp.getParent());
            end if;
      end if;
end function;  // started

function stopped(Component comp):
      if (alternativeComponentAvailable())
            create(comp.getChild(comp.getDependency(), determOption()));
      else  // backtracking necessary
            Contract btContract = findBacktrackContract(); // see Sect. 4.3
            if (btContract == null)
                  terminateUnsuccessfully(); // no suitable contract found
            end if;
            Instance toStop = btContract.getInstance();
            Component toStart = determineAlternativeComp(btContract);
            backtrack(toStop, toStart);
            create(comp.getChild(comp.getDependency(), comp.getOption());
      end if;
end function;  // stopped

function backtrack(Instance toStop, Component toStart):
      stopSubtreeInstances(toStop); stopInstance(toStop);
      startInstance(toStart); startSubtreeInstances(toStart);
      return;
end function;  // backtrack
```

Fig. 3. Direct Backtracking

the number of conflict situations that make a backtracking process inevitable. This fundamental mechanism of DBT is described in detail in Section 4.2.

If there are enough resources available to resolve a dependency within the *create* function, two possibilities exist:

1. The component to be started represents a leaf of the tree. In this case, the *started* function is called subsequently to indicate this instantiation of a leaf component to its parent component.
2. The component to be started represents an inner node (i.e., no leaf of the tree). In this case, the *create* function is called recursively to instantiate the components on the next lower level of the tree.

If not enough resources are available to instantiate a component at the moment, the *stopped* function is called to indicate a resource conflict situation to the parent component. In this case, DBT at first tries to instantiate the next component option. If no further option exists, a *backtrack* process has to be initiated. This function at first stops the adherent subtree of the selected backtracking component C_1, then stops C_1 itself, and instantiates another component C_2 and the previously stopped subtree components. C_2 has been determined before within `findBacktrackContract()` that is further described in Section 4.3. Afterwards, DBT retries to instantiate the component which could not be instantiated previously. The algorithm terminates within the *started* function when the anchor is reached and all dependencies of the anchor have been resolved.

4.2 Proactive Backtracking Avoidance

In case of a multi-optional contract for a certain dependency, the selection of a component has to be made cautiously in order to avoid conflict situations right from the start and reduce the number of situations in which backtracking is necessary. For this purpose, Direct Backtracking contains a proactive mechanism which carefully selects the component option to be instantiated in order to avoid backtracking.

Within multi-optional contracts, options are ordered in a list according to the container ID, i.e. a component on a container with a lower ID has higher priority for the configuration algorithm. If there exist multiple options for a specific dependency on one container, they are additionally ordered according to their resource consumption: The component with least resource requirements has highest priority.

Direct Backtracking performs the following steps on the ordered list of options in the given order:

1. Initially, DBT selects the first component option in the list to be instantiated, i.e. the highest-priority component.
2. If the currently selected option consumes the total free amount of a resource R, the algorithm scans the ordered list of options for alternative components. If there exists a component on another container whose instantiation would still leave some amount of R unused on this container, DBT adjusts the selected option in order to decrease the potential for future conflicts. This adjustment of the option to be instantiated is performed to the highest-prior component that fulfills the above condition of leaving some amount of R unused.
3. For the currently selected option, DBT verifies that there are enough global resources remaining to fulfill all missing contracts in theory after the initialization of the chosen component C. This means that there have to exist at least i free components among **all** devices with sufficient resources to fulfill the i dependencies of the application that have not been resolved yet. Otherwise, C is not instantiated at this moment as this would yield a future inevitable backtrack process. In this case, the algorithm selects the next lower-prior option in the ordered list and continues with step 2.

If none of the options can be instantiated, a backtracking is necessary. DBT's intelligent backtracking process is described in the next subsection.

4.3 Intelligent Backtracking

In case of a conflict situation, if none of the possible components for a contract C could be instantiated, a backtracking has to be initiated. Thus, another contract that can be adapted must be found. In many situations, there is more than just one candidate for an adaptation. In such a case, DBT performs an intelligent backtracking by carefully selecting a contract whose components can be adapted with little overhead.

First, let us assume that the components of C which could not be instantiated due to a shortage of a specific resource R form a set of components $S_1 = \{Cmp_1, Cmp_2, ..., Cmp_i\}$. If C is *not* multi-optional, S_1 includes only one component. Now, DBT determines the set $D_s = \{D_1, D_2, ..., D_j\}$ of containers which host at least one component that is included in S_1, i.e.

$$(\exists k, j : k \in \{1, ..., i\} \wedge l \in \{1, ..., j\} \wedge Cmp_k \in D_l) \Rightarrow D_l \in D_s \qquad (1)$$

Subsequently, DBT determines another set, S_2. This set contains those multi-optional contracts for which a component is currently instantiated that is resident on one of the containers included in D_s. Furthermore, only those contracts are included in S_2 for which an alternative component on another container Ctr exists that can be instantiated *now* due to sufficient uninstantiated amount of R on Ctr. The contracts in S_2 are ordered in descending order according to the amount of R that is consumed by the instantiated component. This means that the instantiated component which consumes the largest amount of R is at the beginning of the list because its termination would cause a considerable deallocation of resources. This helps to decrease the number of needless adaptations which would have to be revised later. If only one suitable component exists, the backtracking target is found and the adaptation process can be initiated.

In case of more than one suitable backtracking targets that consume an identical amount of R, an additional selection criterion is necessary for weighting them according to their suitability for adaptation. Since adaptation is simpler for contracts with small adherent subtrees (as the subtree also has to be adapted), DBT selects the component C that has least descendants (number of all child components down to the leaves) and, hence, is closest to the bottom of the tree. Thus, contracts with little adaptation overhead are preferred. In case of multiple contracts with a subtree of the same size, the algorithm selects the one with the highest priority, i.e. the one with lowest index.

If the resource conflict cannot be solved by adapting the first contract in S_2, DBT tries to solve it by adapting the second contract in S_2, and so on. If the conflict cannot be solved by adapting any contract included in S_2, this indicates that there are not sufficient resources in the environment. Thus, the algorithm terminates unsuccessfully within the *stopped* function and informs the user of this failure.

4.4 Example for Adaptation Process

Now let us revisit the example presented in Section 2.1 to see how the intelligent backtracking mechanism of DBT performs in practice. In the situation depicted in Figure 2, Direct Backtracking recognizes that the conflict arises for R_3. S_1 consists of the two possible components of C_6. These components are resident on D_A and D_B. Thus, we have a set $D_s = \{D_A, D_B\}$. Both of these devices lack one instance of resource R_3. According to the procedure described in Section 4.3, DBT identifies C_1 and C_3 as possible contracts for adaptation. The currently instantiated components of C_1 and C_3 both have allocated an identical amount of one instance of R_3. Hence, the subtree criterion has to be taken into account and C_3 is elected for adaptation because it has no adherent subtree which would cause additional adaptation effort. Thus, DBT directly backtracks to C_3, stops the instantiated component on D_B, instantiates the component on D_A, and returns to C_5. Now, DBT is able to instantiate the component on D_B to fulfill C_6, as sufficient free resources are available. Subsequently, the entire application is successfully instantiated.

In the same situation, Synchronous Backtracking would at first try to adapt contract C_4 which is the next higher-priority multi-optional contract above C_6. But since C_4 depends on a different resource, this adaptation of C_4 does not solve the problem. So, SBT has to initiate another backtracking process which would adapt C_3 as the next higher-priority multi-optional contract. This second adaptation would resolve the conflict.

Since SBT always performs backtracking stepwise to the previous contract in the tree, and because of the useless adaptation of C_4, the backtracking function is executed multiple times, while for DBT, only one direct backtrack to C_3 is necessary. Hence, due to its intelligent backtracking mechanism, DBT performs adaptation of this exemplary application with less overhead and, thus, much faster than SBT does. As DBT additionally avoids backtracking processes in many situations due to the proactive mechanism described in Section 4.2, DBT's benefit even increases in many resource-constricted environments.

5 Evaluation

This section presents our evaluation results. We simulated the algorithm on a discrete event PCOM simulator.

We compared Direct Backtracking to Synchronous Backtracking (SBT), the centralized version of Asynchronous Backtracking [7] that was used previously for distributed application configuration [5]. We chose to compare DBT to SBT as this is the best centralized algorithm that is applicable to the problem of adapting pervasive applications without excessive memory waste and computation overhead. As already mentioned in the related work section, a comparison of Direct Backtracking to Dynamic Backtracking (DyBT) is not possible since DyBT changes the order of the contracts to resolve conflicts, which is not an option here.

5.1 Experimental Setup

The simulated environments are constructed as follows to support various different scenarios: We create an application that consists of n instances by adding n components to a binary application tree. Then, we create one container and place the anchor on it. For the remaining (n-1) components, we create $m \leq n$ containers and place the components on those containers in a round robin manner. Then, we artificially create conflicts by replicating k random components and by increasing their resource needs to two without increasing the available resources on the containers. Thus, increasing k will lead to a higher potential for conflicting selections during automatic configuration as it increases the number of multi-optional contracts.

Now, we varied the height h of the binary tree from two, where an application consisted of only $2^3 - 2 = 6$ components (excluding the anchor component), up to ten, which leads to $2^{11} - 2 = 2046$ components. For each tree height, we varied the number k of replicated components and ran 10000 simulations.

Regarding real-life pervasive scenarios, environments typically exist of many different devices with distinguished functionalities. They can be homogeneous as well as heterogeneous. Frequently, devices with similar or even the same functionality are available. This means that applications which use this functionality also include multi-optional contracts.

Hence, it can be seen that the assumed simulated setting is realistic for pervasive computing environments.

5.2 Evaluation Results

We evaluated the algorithm on a common desktop PC[2] and compared our algorithm to Synchronous Backtracking, especially concerning the aspects of *configuration latency*, *communication overhead*, *success quota*, *memory overhead* and *size of the source code* and got following results:

- **Configuration latency:** The main goal in the development of an advanced centralized configuration algorithm was the reduction of the configuration latency that is noticeable for the user. Figure 4 shows DBT's latency for three different heights of the binary tree and, thus, different application sizes. The graph shows the latency relative to the respective SBT performance. Thus, values above 1.0 indicate a DBT performance that is worse than that of SBT, while values below 1.0 indicate better performance. It can be seen that especially for a huge number of multi-optional contracts, there is an immense improvement when using DBT since it performs proactive backtracking avoidance, intelligent backtracking, and avoids thrashing by considering the cause of a backtrack. The figure also shows that the relative performance of DBT even increases with increasing application size. According to this figure, DBT induces just about 4.2 % of the latency of SBT if an application consists of 62 contracts, whereas 14 contracts are multi-optional.

[2] PC with Dual Core Processor (2.2 GHz), 2.0 GB of RAM

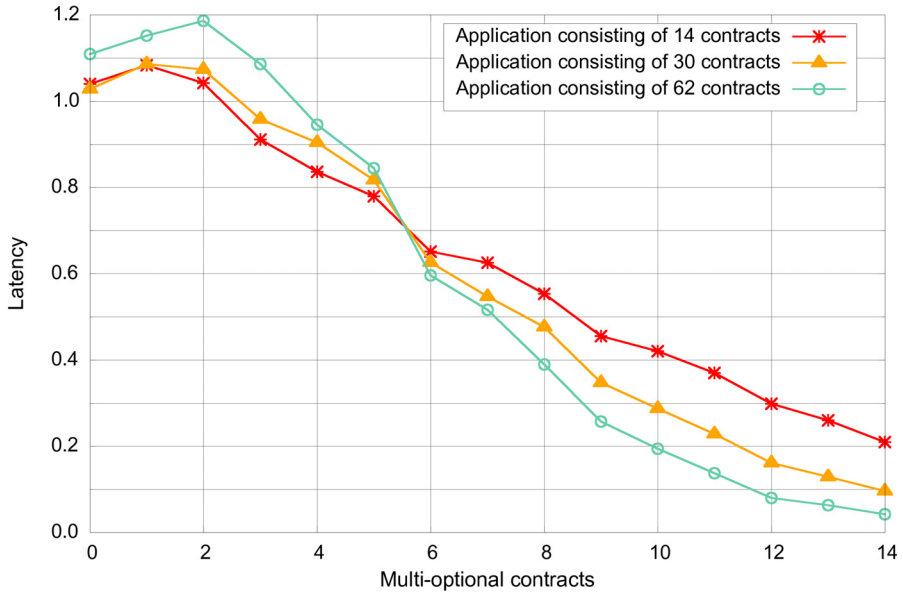

Fig. 4. Configuration latency of DBT (compared to SBT references)

Nevertheless, it must be mentioned that for small fractions of multi-optional contracts, SBT performs better than DBT by up to 20 %. This is because of DBT's additional checks for avoiding conflict situations and the process of storing the backtracking causes. Since absolute latencies were very small in those cases (in the range of few milliseconds), this overhead can be neglected. With increasing tree height, the *crossover point*, which represents the fraction of multi-optional contracts for which DBT starts to outperform SBT, exponentially decreases, as it can be seen in Figure 5. This means that for huge applications, even if just a small amount of contracts is multi-optional, DBT is the better choice regarding configuration latency.

- **Communication overhead:** Compared to the other centralized backtracking algorithms presented in Section 3, no additional communication overhead arises during runtime of our algorithm.
- **Success quota:** The scenarios have been created in a way that at least one valid configuration exists for each scenario. In every single simulation run, both SBT and DBT terminated successfully with a valid application configuration.
- **Memory overhead:** Compared to SBT, DBT needs to store additional information about arising conflicts, especially the contract chosen for adaptation and the contract to which the algorithm has to return after an adaptation has been performed. We measured the average random access memory consumption of the algorithm on a common desktop PC and compared it to SBT. While memory consumption of SBT was almost independent from

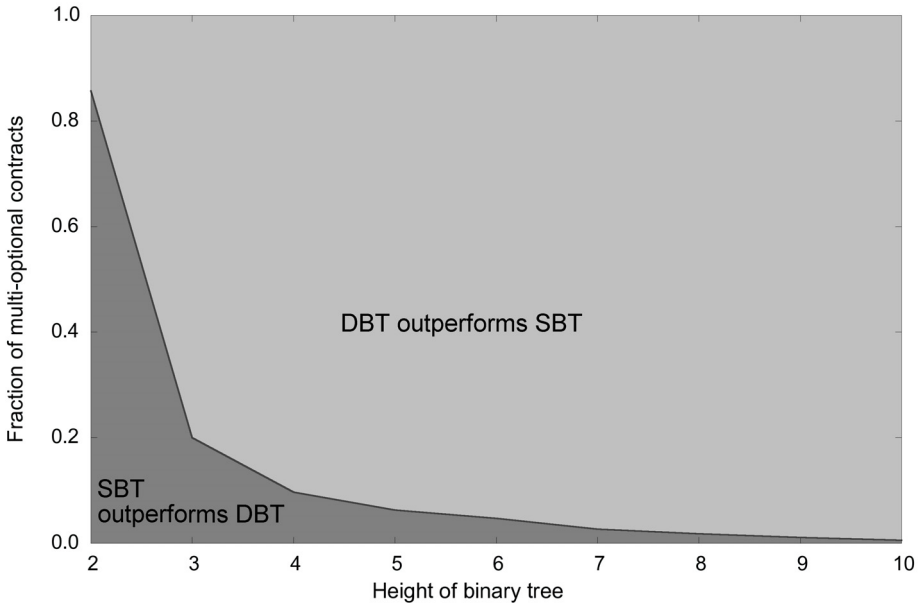

Fig. 5. Crossover points

the application size and the number of conflicts (the standard deviation was below 2 % in all runs), the overhead of DBT increased with the application size and the number of multi-optional contracts, but remained within acceptable limits. The average memory overhead of DBT varies between 8.0 % for applications with 6 components and 27.6 % for large applications with 2046 components. The maximum overhead of DBT compared to SBT in a single simulation run was 38.5 %, the absolute amount of required memory was 19.2 Megabytes. The results of the memory evaluation are shown in Figure 6.

– **Source code size:** While our Synchronous Backtracking implementation needs 10.8 kB of disk space, Direct Backtracking consumes about 96.9 kB.

Regarding that the algorithm is optimized for use on resource-rich devices because of its centralized nature, the additional code overhead as well as the memory overhead do not prevent the use of DBT.

6 Conclusions and Future Work

We have presented Direct Backtracking (DBT), a new centralized algorithm for efficient configuration and adaptation of tree-based pervasive applications. Our approach avoids thrashing completely due to an intelligent backtracking mechanism, while memory and code overhead are of acceptable size. Furthermore,

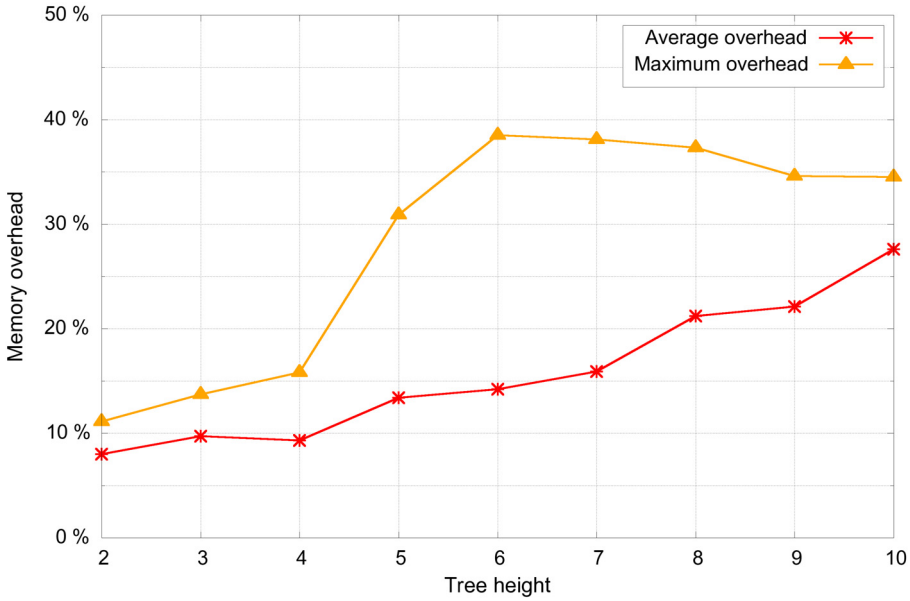

Fig. 6. Memory overhead of Direct Backtracking

DBT avoids adaptations in many situations as it employs a proactive backtracking avoidance mechanism.

We have shown that DBT significantly outperforms Synchronous Backtracking (SBT) for applications with various sizes, and especially for huge scenarios with many multi-optional components where it causes less than 5 % of SBT's configuration latency. The small latency overhead of up to 20 % for scenarios with a low fraction of multi-optional contracts can be neglected since absolute latencies for these applications were very small. The additional memory and code overhead of DBT is not significant as the algorithm is designed for centralized configuration on resource-rich devices.

Our future work concentrates on the efficient support of hybrid configuration and adaptation in homogeneous as well as heterogeneous environments by introducing a clustering scheme. Within each cluster, the cluster heads configure applications in a centralized way, while the different cluster heads perform distributed configuration among each other. Direct Backtracking will be used as efficient centralized algorithm for configuration and adaptation on the cluster heads.

7 Acknowledges

This work is funded by the German Research Foundation within DFG Priority Programme 1140 - Middleware for Self-organizing Infrastructures in Networked Mobile Systems.

References

1. C. Becker, M. Handte, G. Schiele and K. Rothermel: *PCOM - A Component System for Pervasive Computing.* Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom 2004), Orlando, USA, 2004.
2. M. Román, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell and K. Nahrstedt: *Gaia: A Middleware Infrastructure to Enable Active Spaces.* In IEEE Pervasive Computing, pp. 74–83, October-December 2002.
3. J. P. Sousa and D. Garlan: *Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments.* Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture, Kluwer Academic Publishers, pp. 29–43, August 2002.
4. U. Saif, H. Pham, J. M. Paluska, J. Waterman, C. Terman and S. Ward: *A case for goal-oriented programming semantics.* UbiSys '03: Workshop on System Support for Ubiquitous Computing at UbiComp '03, Seattle, USA, pp. 1–8, 2003.
5. M. Handte, C. Becker and K. Rothermel: *Peer-based Automatic Configuration of Pervasive Applications.* IEEE International Conference on Pervasive Services 2005 (ICPS'05), Santorini, Greece, 2005.
6. M. Handte, S. Urbanski, C. Becker, P. Reinhard, M. Engel and M. Smith: *3PC/MarNET Pervasive Presenter.* 4th IEEE International Conference on Pervasive Computing and Communications (PerCom 2006), Pisa, Italy, 2006.
7. M. Yokoo, E. H. Durfee, T. Ishida and K. Kuwabara: *The Distributed Constraint Satisfaction Problem: Formalization and Algorithms.* IEEE Transactions on Knowledge and Data Engineering, 10(5), pp. 673–685, 1998.
8. A. B. Baker: *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results.* PhD Thesis, University of Oregon, March 1995.
9. I. Brito and P. Meseguer: *Synchronous, asynchronous and hybrid algorithms for DisCSP.* Workshop on Distributed Constraints Reasoning (DCR-04), Toronto, Canada, September 2004.
10. J. Gaschnig: *A general backtrack algorithm that eliminates most redundant checks.* International Joint Conference on Artificial Intelligence, Menlo Park, p. 457, 1977.
11. R. M. Stallman and G. J. Sussman: *Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis.* Artificial Intelligence, 9, pp. 135–196, 1977.
12. M. L. Ginsberg: *Dynamic backtracking.* Journal of Artificial Intelligence Research, 1, pp. 25–46, 1993.