AUTOMATED WAVEFORM PARTITIONING AND OPTIMIZATION FOR SCA WAVEFORMS

James Neel (Mobile and Portable Radio Research Group, Wireless @ Virginia Tech, Blacksburg, VA, USA; janeel@vt.edu); Carlos Aguayo (MPRG, Wireless @ Virginia Tech); Jeffrey Reed (MPRG, Wireless @ Virginia Tech).

ABSTRACT

Waveform partitioning – the process of assigning waveform components to processors – is a key step in the design of waveforms for implementation on multi-processor systems. For software radios employing processor pooling, waveform partitioning is a recurring task which needs to be performed in near real-time to support the run-time addition and removal of waveforms. This paper presents three different algorithms for automating the waveform partitioning process, compares the performance of these algorithms, and describes how partitioning algorithms can be incorporated into an SCA-compliant software radio. Details are given on how these partitioning algorithms are being integrated into OSSIE – the open source SCA implementation for embedded systems hosted by Virginia Tech.

1. INTRODUCTION

Software radios are frequently implemented on multiple processor platforms necessitating the partitioning of waveform components across the processors. For single waveform radios, this partitioning can be pre-computed. However, many JTRS radios and SDR base stations are intended to support multiple simultaneous channels possibly with the different waveforms operating on different channels. If the radio is designed so each channel has its own dedicated processing hardware, then the partitioning solution can again be pre-computed. But dedicating processing hardware to each channel is less efficient than allocating resources from a pool of processing elements [1]. With a multiple-channel multiple-waveform software radio implemented on a dynamic pool of processing elements, pre-computation of partitions is not feasible and a dynamic partitioning solution should be adopted.

Fortunately, the Software Communications Architecture (SCA) was designed to support dynamic partitioning of waveform components over modular platforms where the number and characteristics of processing elements can change over time. The *ApplicationFactory* interface, part of the SCA Core Framework, is intended to collect information about the underlying platform and the processing requirements of the waveform components. A routine can use this information to guide the allocation of processing resources and partition waveforms so that waveform components are assigned in such a way to minimize power consumption or maximize some other design objective.

The problem of solving for the optimal assignment of components to devices is analogous to a knapsack problem – a classic integer programming problem. Traditionally, in a knapsack problem, hikers must pack out a number of different items of various values and sizes by assigning items to the knapsacks where different knapsacks have different capacities. The objective of the problem is to find the allocation of items that maximizes the total value as constrained by capacities of the knapsacks. For the purposes of waveform partitioning in a processor pool, each processor is a “knapsack” and each waveform component is an “item” whose “size” is the resources consumed by the component (e.g., cycles or CLBs). While the equivalence between the waveform partitioning problem and the knapsack problem implies that solving for an optimal partition is unfortunately an NP-complete problem, it also means that we can draw on existing operations research literature which has studied knapsack related problems (e.g., [2] and [3]) to identify candidate algorithms for waveform partitioning. However, this problem differs from the knapsack problem due to the existence of multiple constraints and that the “size” of the “items” vary by “knapsack”.

This paper presents algorithms for dynamically partitioning SCA waveforms on hardware platforms utilizing processor pooling, describes how the algorithms are incorporated into Virginia Tech’s open source SCA implementation – OSSIE [4], and presents the results of simulation studies into the performance of the candidate partitioning algorithms.

2. WAVEFORM PARTITIONING

In this formulation, the waveform partitioner is tasked with finding the placement of waveform components that optimizes some parameter such as minimizing power consumption while satisfying the processing requirements.

2.1 Formal Description of Waveform Partitioning

To formalize this problem, we introduce some terminology. We say that a waveform, *w*, consists of a set *C* of waveform components where each component $c \in C$ is to be implemented on some device $d \in D$ where *D* is the set of devices (processors) available in the processor pool. A *solution*, ***s***, is an association of each component with a device. For programming purposes a solution can be treated as a vector of length |*C*| (the number of components) where entry s_j specifies the device on which component c_j is to be implemented. For instance, a partitioning solution, *s*, of a seven component waveform with components c_0 to c_7 onto a two device platform, $D=\{d_1, d_2\}$ might be specified as $s=(d_1, d_2, d_2, d_2, d_1, d_2, d_1)$ where *s* is actually one of $|D|^{|C|}$ possible solutions when every component has an available implementation for every device (there are 2^7 possible solutions in this example).

However, a radio will typically not have the requisite code or bit image to implement every component on every device. To model this reality, we say that a component *c* can be implemented on the set *D*(*c*) of devices. So there are actually only $\underset{c_i \in C}{\times} |D(c_i)|$ possible solutions to search through. Again, not all of these implementations will be feasible as there may not be sufficient resources available on a device to satisfy the processing demands of all of the components specified as part of a partitioning solution.

We can capture this feasibility limitation as follows. If we denote the memory requirements of component *c* on device *d* as *mem*(*c*, *d*), the cycle requirements as *cycle*(*c*, *d*), and the processing element requirements as *elements*(*c*, *d*), a solution **s** is not feasible if any of (1), (2), or (3) is violated where max(*d*, *mem*), max(*d*, *cycles*), max(*d*, *elements*) are respectively the maximum amount of memory, cycles, and processing fabric elements available on device *d*.

$$\sum_{s_i=d} mem(c_i, d) < \max(d, mem) \tag{1}$$

$$\sum_{s_i=d} cycles(c_i, d) < \max(d, cycles) \tag{2}$$

$$\sum_{s_i=d} elements(c_i, d) < \max(d, elements) \tag{3}$$

In a processor pooling scheme, max(*d*, *mem*), max(*d*, *cycles*), and max(*d*, *elements*) are a function of any currently operating waveforms and may need to be adjusted to provide a safety margin for statistical resource contentions. With these constraints in place, and assuming we are looking for the solution **s** that maximizes some objective *O*(**s**) (e.g., power minimization), we can represent the partitioning problem as an integer programming problem as follows.

$$\begin{array}{ll} \text{Maximize:} & O(\mathbf{s}) \\ \text{Subject to:} & (1),(2),(3) \\ & s(c_j) \in D(c_i) \;\; \forall c_j \in C \end{array} \tag{4}$$

2.2 Techniques for Solving the Waveform Partitioning Problem

Unfortunately, (4) cannot be solved as a simple system of equations, so we must employ search algorithms in a hopefully intelligent version of trial and error. This search process can be quite daunting as the analogous knapsack problem (or general assignment problem) is known to be an NP-complete problem and simply determining that a feasible solution exists is also an NP-complete problem. Further it is known that no polynomial-time algorithm with a fixed worst-case performance ratio (e.g., 50% of the optimum value) exists [2]. Thus the design of our partitioning algorithm is faced with a tradeoff between performance and run-time. Using the terminology presented in Section 2.1, the following presents three algorithms which can be used to generate solutions to the waveform partitioning problem.

In an exhaustive search, every possible solution is tried, evaluated for feasibility, and evaluated for optimality. This can be a very time consuming process as all $\underset{c_i \in C}{\times} |D(c_i)|$ are tried in the process, but an exhaustive search is guaranteed to find the optimal feasible partition if one exists. Referring to *S* as the set of solutions and numbering the solutions as s^1 to $s^{|S|}$, the exhaustive search algorithm can be written as shown in Figure 1.

```
o* = -∞
for k = 1:|S|
   if $s^k$ is feasible
      temp_o = $O(s^k)$;
      if temp_o > o*
         sol = $s^k$;
         o* = temp_o;
      end
   end
end
```

Figure 1: Exhaustive Search Algorithm

In a local search algorithm, all solutions within the feasible neighborhood of the current solution, s^k, $N(s^k)$ are evaluated for feasibility and performance. The next solution, s^{k+1} is found as $s^{k+1} = \arg\max_{s \in N(s^k)} O(s)$. For the purposes of this algorithm, we define the feasible neighborhood of *s* as the union of *s* with the set of feasible solutions which only differ from *s* in the assignment of a single component. The local search algorithm terminates when $s^{k+1} = s^k$.

```
$max\_val = O(s^0)$;
$continue$ = **true**;
$k$=0;
**while**($continue$)
    $s^{k+1} = s^k$;
    $continue$ = **false**;
    **for** $j$=1: $|C|$
        **for** $m$ = 1:$|D(c^j)|$
            $temp\_s = \left(s_0^k,\ldots,d_m \in D\left(c_j\right),\ldots,s_{|C|}^k\right)$
            **if** $temp\_s$ is feasible **and** $O(temp\_s) > max\_val$
                $continue$ = **true**;
                $max\_val = o(temp\_s)$;
                $s^{k+1} = temp\_s$;
            **end**
        **end**
    **end**
    $k$=$k$+1;
**end**
```

Figure 2: Local Search Algorithm

In general, the local search algorithm converges faster than the exhaustive search, but is not guaranteed to converge to an optimal solution nor a feasible solution with the exact solution a function of the initial solution. To overcome this dependence on initial conditions, local search algorithms are frequently randomly restarted with different initial solutions.

The greedy algorithm presented in Figure 3 is based on an approximation presented in Section 7.4 in [1] where we only changed the meaning of the symbols and the number of constraint equations – from just "weight" to memory, cycles, and processing elements. In $o(c,d)$ is the added value of assigning component c to device d. For example if minimizing total power consumption, $o(c,d)$ is the negation of the amount of power c consumes when implemented on d.

Beginning with all components unassigned, the greedy algorithm iteratively considers all unassigned components and finds the component, c^*, which has the biggest difference between the largest and second largest feasible $o(c,d)$ for all $d \in D(c)$. The component c^* is then assigned to the device that maximizes $o(c^*,d)$. The algorithm continues until all components are assigned or feasible assignments are impossible. While this algorithm completes in polynomial time, like the local search, this greedy algorithm is not guaranteed to find a feasible solution or the optimal solution.

3. AUTOMATING WAVEFORM PARTITIONING IN SCA COMPLIANT RADIOS

In an SCA system there are two different classes of components – *Devices* that represent hardware elements and *Resources* that perform signal processing. The former are used to abstract platforms while the latter are used to build waveforms. Remember that in the SCA context, a waveform is the set of transformations applied to information that is transmitted over the air and the corresponding set of transformations to convert received signals back to their information content [5]. In other words, an SCA waveform is the set of software components and interconnections required to implement a particular wireless protocol.

```
$\bar{C} = C$;
$feas$ = **true**
**while**($feas$ and $\bar{C} \neq \varnothing$)
    $diff^*$ = $-\infty$;
    **for each** $c_j \in \left|\bar{C}\right|$
        $F_j = \{\, d \in D\left(c_j\right):\ mem\left(c_j,d\right) < \max\left(d,mem\right)$,
         $cycles\left(c_j,d\right) < \max\left(d,cycles\right)$, and
         $elements\left(c_j,d\right) < \max\left(d,elements\right) \}$;
        **if** $F_j \neq \varnothing$, **then** $feas$ = **false**
        **else**
                $d_j^* = \arg\max_{d \in F_j} o\left(c_j,d\right)$;
                $F_j^* = F_j \setminus d_j^*$;
                **if** $F_j^* = \varnothing$, **then** $diff = \infty$;
                **else**
                    $d_j^{**} = \arg\max_{d \in F_j^*} o\left(c_j,d\right)$;
                    $diff = o\left(c_j,d^*\right) - o\left(c_j,d^{**}\right)$;
                **end**
                **if** $diff > diff^*$
                    $diff^* = diff$;
                    $d^* = d_j^*$
                    $c^* = c_j$
                **end**
        **end**
    **end**
    **if** ($feas$)
        $\bar{C} = \bar{C} \setminus c^*$;
        $s(c) = d^*$;
        $\max\left(d^*,mem\right) -= mem\left(c^*,d^*\right)$
        $\max\left(d^*,cycles\right) -= cycles\left(c^*,d^*\right)$
        $\max\left(d^*,elements\right) -= elements\left(c^*,d^*\right)$
    **end**
**end**
```

Figure 3: Greedy Assignment Algorithm

SCA waveforms achieve a degree of platform independence via the operating environment provided by the SCA. This operating environment consists of a POSIX-compliant Operating System, CORBA middleware, and the SCA Core Framework (CF). The CF provides a set of interfaces to deploy, manage, and configure SCA waveforms and platforms. It describes a set of required interfaces for *Devices*, *Resources* and generic management artifacts. It also describes a Domain Profile, a set of XML descriptors that provide machine readable information about waveforms, resources, and platforms.

The SCA provides a flexible architecture where every waveform could run on any platform, and every platform could support any waveform in theory. However, this flexibility adds complexity to the management of a radio. To help cope with this complexity, several mechanisms based on well-known software design patterns have been integrated into the SCA. Of interest for our waveform partitioning problem, the SCA relies on an *ApplicationFactory* to instantiate waveforms.

3.1 SCA *Resources*

Given the wide variety of possible hardware configurations, software resources may significantly improve their performance by utilizing specific features of a particular platform, e.g., a convolutional decoder might be implemented on a hardware accelerator. To take advantage of these features, the SCA allows *Resources* to have multiple implementations to provide the same functionality. For example, a software component may have a general implementation for a GPP, and a specialized one that runs on a particular DSP. Implementation tradeoffs for the same platform are also possible. If a platform has plenty of memory, an implementation may be able to unroll loops or utilize a table implementation, improving performance at the expense of memory utilization. Capacity requirements may be different for every implementation of a *Resource*.

Resources and their capacity requirements are described in the Domain Profile. Each component description includes a Software Package Descriptor (SPD), a Software Component Descriptor (SCD), and an optional Property File (PRF). The SPD file contains a list with all of the available implementations for the component. Each implementation may have a reference to a PRF file that describes the particular allocation requirements.

3.2 SCA Devices

A *Device* is a special kind of *Resource* that provides a logical abstraction of a physical device. It acts as a proxy for the rest of the components to interact with the piece of hardware it represents. Because of the wide variety of physical devices that can be integrated into a platform, the SCA allows each *Device* to define its own capacity model. That is, each device can specify what services it can provide, how these services are quantified, and the way it provides this services. This capacity model is described by means of allocation properties and is component dependent. For example, a microcontroller in a platform may indicate that it has available 32 MB of free memory to host software components; an FPGA, that it has 3 MBLPs available; a DSP, that it has 32 MB of memory available, 500M idle clock cycles, one available McBSP interface, and a dedicated convolutional decoder coprocessor.

Before deploying any application, the *ApplicationFactory* obtains a list of all *Devices* available for the platform and their respective capacities. This information is contained in the Device Profile. Each *Device* is described by a Device Package Descriptor (DPD) and a SPD. The former describes the hardware part while the latter describes the logical part of that device. The SPD may also reference a PRF which describes the properties of the device being deployed such as serial number, processor type, and allocation capacities. A Device Configuration Descriptor (DCD) provides a list of all the devices deployed at startup.

3.3 Dynamic Partitioning in *ApplicationFactory*

The SCA *ApplicationFactory* is based on the Factory Design Pattern. It partitions, deploys, and instantiates waveforms based on information provided by the *Software Assembly Descriptor* (SAD). This descriptor contains a list of all the components that comprise the waveform and their respective connections. *ApplicationFactory* is expected to find an appropriate device on the platform to host each component in the waveform. This decision is based on each *Device*’s capacity model and the requirements and dependencies of the software components, e.g., a software component developed for a DSP would not be able to run on an ARM. Some components may require a particular hardware accelerator or a particular amount of memory – all of which must be considered by the *ApplicationFactory* when partitioning a waveform. Besides these obvious matches, *ApplicationFactory* needs to include the developer’s input for a predefined deployment plan and collocation requirements for some components. Collocation requirements refer to those components that need to be deployed on the same device for proper operation, or to meet performance constraints.

It is at this point that *ApplicationFactory* executes the algorithms described in this paper to find the best partitioning solution according to a predefined optimization criterion. This criterion could be smaller memory footprint, faster performance, lower power consumption, or fewer number of devices required. After the solution has been found, *ApplicationFactory* allocates the required capacities from the host devices and proceeds to launch, initialize, configure, and connect components.

3.4 Integration into OSSIE

The Open-Source SCA Implementation::Embedded (OSSIE) is an open implementation of the SCA, developed by Mobile and Portable Radio Research Group (MPRG) at

Virginia Tech. It is developed in C++, runs on a Linux platform, and uses omniORB and the Xerces XML parsers. To validate the feasibility of automated waveform partitioning and implementation, we developed a sample implementation of the algorithms presented in this paper. They are implemented as separate classes and invoked from the OSSIE framework. The OSSIE *ApplicationFactory* obtains the description of the platform and waveforms from the XML domain profile. It then writes an intermediate file containing all the available devices and requested resources along with their respective allocation capacities. The partitioning classes read this file, find the best partitioning solution, and write another intermediate file containing the component-device pairs that represent the deployment plan. The OSSIE framework reads this output file and converts it into a *DeviceAssignmentSequence* which is used for waveform creation. Prior to this work, the OSSIE framework required an externally supplied *DeviceAssignmentSequence*.

4. PERFORMANCE

To inform the choice of algorithms for the partitioner, we developed a simulation of the waveform partitioning process on a pooled processing system. The simulation permits us to vary the number and type of devices available in the system, the number and type of components in the target waveform, and the amount of resources utilized by concurrent and previously allocated waveforms.

We drove this simulation with varying number of devices modeled on the Texas Instruments TMS320VC5501. To model previously allocated waveforms, the available resources for each device were reduced by a random number drawn from a uniform distribution from zero to the device capacity. Each algorithm was then tasked with partitioning a waveform whose components have randomly generated processing requirements with the goal of minimizing total power consumption. This process was repeated over ten trials for each combinations of numbers of components and devices.

The results of these simulations are shown in Figure 4 where the performance of an algorithm is measured as fraction of the performance of the exhaustive search algorithm where achieving the same power level as the exhaustive search is assigned a 1, not finding a feasible solution when one exists a 0, and all other values are given as power(exhaustive search)/power(algorithm). Because power is modeled as a linear function (P = cycles/max_cycles $\times$ Peak_power), all feasible solutions consume the same power so the plots effectively depict feasibility percentages.

To examine the performance for the more general case of a heterogeneous processing pool, we used two different device types – the VC5501s from before and Blackfins operating in their peak power consumption modes. The results of these simulations are shown in Figure 5 where relative local search performance is given as was done in Figure 4, but with the greedy algorithm in the place of exhaustive search. Unlike the previous simulation, not every feasible solution is an optimal solution, and the ratio of

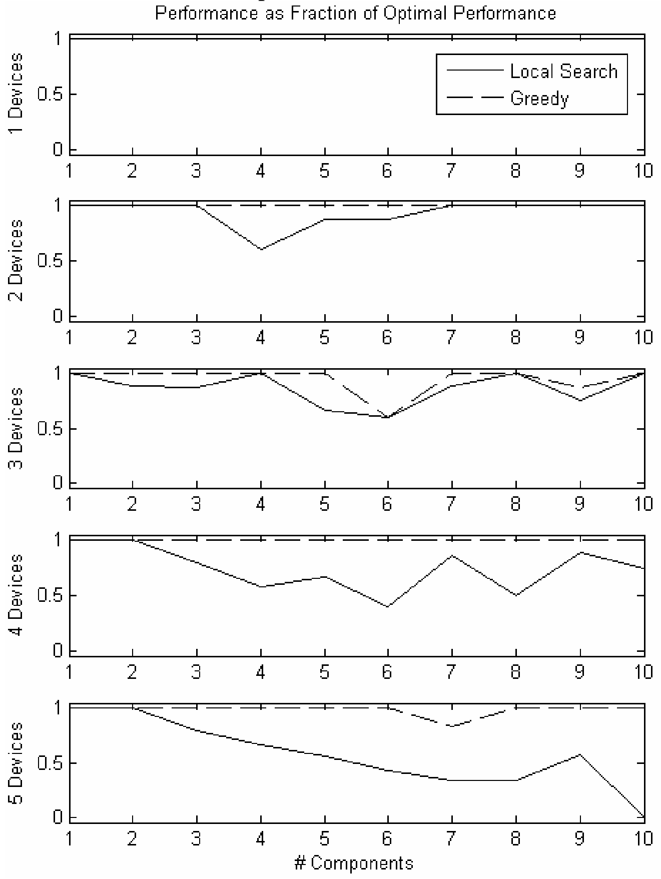

Figure 4: Average performance of partitioning algorithms with respect to exhaustive search in uniform processor pool.

power consumption of the greedy result to the local search result could exceed 1 implying that the local search algorithm outperformed the greedy search. However, out of the 1200 individual trials in this simulation, the local search outperformed the greedy algorithm only twice. As a real-world comparison, an OFDM waveform with 10 MHz channels developed by Nova Systems Solutions and intended for deployment on high-end platforms has nine components implemented over two different processors (a TMS3206416 and a Virtex 2V6000-6).[6]

For this more extensive simulation, we did not perform the exhaustive search algorithm as it takes significantly longer to execute than either the greedy or local search algorithms as illustrated in Figure 6. More generally for the two deterministic algorithms (the local search was permitted to randomly restart up to 10 times if it failed to find a feasible solution), the time complexity of the exhaustive search algorithm is $O\left(|C|^{|D|}\right)$ while the greedy algorithm – assuming an initial insertion sort of power consumption by component for each device – is $O\left(|D||C|\log|C|+|D|^2\right)$ [2].

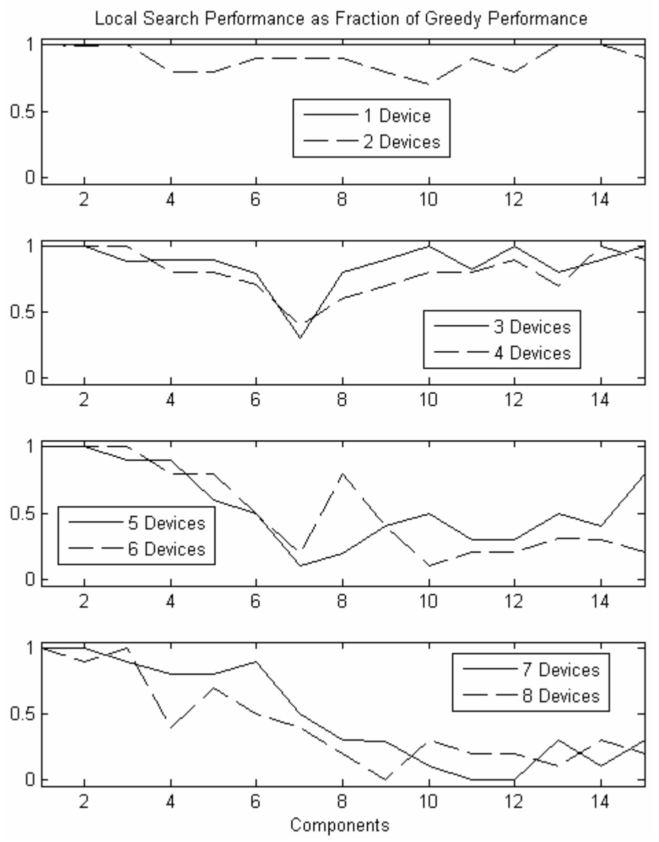

Figure 5: Average performance of local search algorithms with respect to exhaustive search in uniform processor pool.

5. CONCLUSIONS AND FUTURE WORK

By drawing an analogy between knapsack problems and waveform partitioning we were able to leverage existing operations research literature to develop a candidate algorithm – the greedy algorithm – for waveform partitioning for run-time and off-line waveform partitioning for processor pools as it consistently finds near optimal solutions in under 100 ms even for large systems of processors and complex waveforms. With this algorithm and the existing services of the SCA run time waveform partitioning for pooled processors should be feasible

Nominally, many of the problems associated with distribution of waveform processing across numerous processors are handled by the ORB services in the SCA. However, the arbitrary distribution of components on a platform that includes FPGAs, and to a lesser extent DSPs, is currently problematic for these services. Also, while the discussed search algorithms support arbitrary mapping of components to arbitrary processor types, a bit image for an FPGA based component implementation is targeted to a specific location on the FPGA [7] which violates assumptions inherent to the search algorithms.

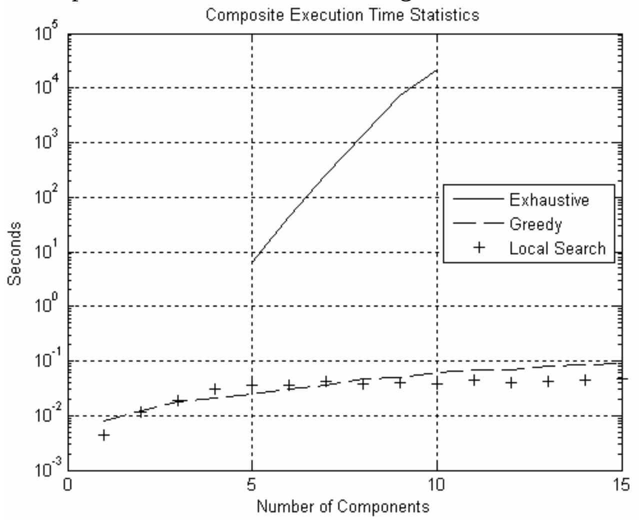

Figure 6: Average **a**lgorithm run times for simulations in Figures 4 and 5.

In the coming months, we will extend this work by more tightly integrating the interface between the search algorithms and OSSIE and modifying processor constraints to support an arbitrary number and type of allocation properties. Additionally, to fully leverage the waveform partitioner, OSSIE will be enhanced to support multiple implementations of components at a framework level and to verify the satisfaction of component dependencies and collocation requirements – features not supported in the current version of OSSIE (version 0.5). Additionally, we will be examining and comparing additional algorithms for solving knapsack problems from [1] and [2] with randomly generated waveform data, externally provided waveform data (e.g., [6]) and waveform data generated in a previous study into the optimal choice of devices for waveforms.

6. REFERENCES

[1] M. Kosmicki, S. Pearce, "Digital Processing Pool for JTRS Software Radio: In-Mission Flexibility and Efficient Technology Insertion," *SDRF Technical Conference 2004*, Phoenix, AZ, Nov. 15-18, 2004.

[2] S. Martello, P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*, John Wiley & Sons Ltd., Chichester England, 1990.

[3] D. Pisinger, "Algorithms for Knapsack Problems," Ph.D. Dissertation, University of Copenhagen, Copenhagen, Denmark, February 1995.

[4] OSSIE website: (OSSIE), http://ossie.mprg.org/

[5] Software Communications Architecture v2.2. Available online: http://jtrs.spawar.navy.mil/sca/

[6] Waveform data provided via correspondence with C. Van der Valk, Nova Systems Engineering.

[7] M. Dumas, L. Belanger, S. Roy, J. Chouinard, **“**Development of a SCA 3.1 Compliant W-CDMA Waveform on DSP/FPGA Specialized Hardware,” *SDRF Technical Conference 2005*, Orange County, CA, Nov 14-18, 2005.