ASCENT: An Algorithmic Technique for Designing Hardware and Software in Tandem

Jules White, Brian Doughtery, and Douglas C. Schmidt
Vanderbilt University, EECS Department, Nashville, TN, USA
Email:{jules, briand, schmidt}@dre.vanderbilt.edu

Abstract—Search-based software engineering is an emerging paradigm that uses automated search algorithms to help designers iteratively find solutions to complicated design problems. For example, when designing a climate monitoring satellite, designers may want to use the minimal amount of computing hardware to reduce weight and cost, while supporting the image processing algorithms running onboard. A key problem in these situations is that the hardware and software design are locked in a tightly-coupled cost-constrained producer/consumer relationship that makes it hard to find a good hardware/software design configuration. Search-based software engineering can be used to apply algorithmic techniques to automate the search for hardware/software designs that maximize the image processing accuracy while respecting cost constraints.

This paper provides the following contributions to research on search-based software engineering: (1) we show how a cost-constrained producer/consumer problem can be modeled as a set of two multidimensional multiple-choice knapsack problems (MMKPs), (2) we present a polynomial-time search-based software engineering technique, called the Allocation-baSed Configuration Exploration Technique (ASCENT), for finding near optimal hardware/software co-design solutions, and (3) we present empirical results showing that ASCENT's solutions average 95%+ of the optimal solution's value.

1 Introduction

Current trends and challenges. Increasing levels of programming abstraction, middleware, and other software advancements have expanded the scale and complexity of software systems that we can develop. At the same time, the ballooning scale and complexity have created a problem where systems are becoming so large that their design and development can no longer be optimized manually. Current large-scale systems can contain an exponential number of potential design configurations and vast numbers of constraints ranging from security to performance requirements. Systems of this scale and complexity—coupled with the increasing importance of non-functional characteristics [9] (such as end-to-end response time)—are making software design processes increasingly expensive [27].

Search-based software engineering [17, 16] is an emerging discipline that aims to decrease the cost of optimizing system design by using algorithmic search techniques, such as genetic algorithms or simulated annealing, to automate the design search. In this paradigm, rather than performing the search manually, designers iteratively produce a design by using a search technique to find designs that optimize a specific system quality while adhering to design constraints. Each time a new design is produced, designers can use the knowledge they have gleaned from the new design solution to craft more precise constraints to guide the next design search. Search-based software engineering has been applied to the design of a number of software engineering aspects, ranging from generating test data [26] to project management and staffing [6, 4] to software security [10].

Open Problem. A common theme in domains where search-based software engineering is applied is that the design solution space is so large and tightly constrained that the time required to find an optimal solution grows at an exponential rate with the problem size. These vast and constrained solutions spaces make it hard for designers to derive good solutions manually. One domain with solution spaces that exhibit these challenging characteristics is hardware/software co-design.

Hardware/software co-design is a process whereby a system's hardware and software are designed at the same-time in order to produce optimizations that would not be possible in either hardware or software alone. Traditionally, hardware/software co-design has focused on determining how to partition application functionality between hardware and software. For example, developers can take a time-critical image processing step in an application and determine whether to implement it in hardware or software. Typically, there are limited resources available to implement functionality in hardware and thus determining which pieces of functionality to implement in hardware versus software becomes extremely challenging. A number of search-based engineering techniques, ranging from particle swarm optimization [1, 29] to genetic algorithms [34, 28, 13] have been used to help automate this process.

This paper examines another type of hardware/software co-design problem that is common in the domain of distributed real-time and embedded (DRE) systems. The problem we focus on is the need to choose a set of hardware and software configuration options that maximizes a specific system capability subject to constraints on cost and the production and consumption of resources, such as RAM, by the hardware and software, respectively. This problem assumes that the partitioning of functionality between hardware and software is fixed but that there are different configuration options in the hardware and software that developers must set.

A configuration option is a setting that can be changed in the hardware or software design, such as the image resolution used in an image processing algorithm. Selecting a higher resolution will yield better image processing results but will in turn require more CPU time. Similarly, the hardware has multiple configuration options, such as the clock speed of the processor used. The goal of engineers is to find a series of settings for these configuration options that maximize a utility function that describes the quality of the system.

For example, when designing a satellite for monitoring earth's magnetosphere [12], the goal may be to maximize the accuracy of the sensor data processing algorithms on the satellite without exceeding the development budget and hardware resources. Ideally, to maximize the capabilities of the system for a given cost, system software and hardware should be configured in tandem to produce a design with a precise fit between hardware capabilities and software resource demands. The more precise the fit, the less budget is expended on excess hardware resource capacity.

A key problem in these design scenarios is that they create a complex cost-constrained producer/consumer problem involving the software and hardware design. The hardware design determines the resources, such as processing power and memory, that are available to the software. Likewise, the hardware consumes a portion of the project budget and thus reduces resources remaining for the software (assuming a fixed budget). The software also consumes a portion of the budget and the resources produced by the hardware configuration. The perceived value of system comes from the attributes of the software design, *e.g.*, image processing accuracy in the satellite example. The intricate dependencies between the hardware and software's production and consumption of resources, cost, and value makes the design solution space so large and complex that finding an optimal and valid design configuration is hard.

Solution approach → Automated Solution Space Exploration. This paper presents a heuristic search-based software engineering technique, called the *Allocation-baSed Configuration Exploration Technique* (ASCENT), for solving cost-constrained hardware/software producer/consumer co-design problems. ASCENT models these co-design problems as two separate knapsack problems [21]. Since knapsack problems are NP-Hard [11], ASCENT uses heuristics to reduce the solution space size and iteratively search for near optimal designs by adjusting the budget allocations to software and hardware. In addition to outputting the best design found, ASCENT also generates a data set representing the trends it discovered in the solution space.

A key attribute of the ASCENT technique is that, in the process of solving, it generates a large number of optimal design configurations that present a wide view of the trends and patterns in a system's design solution space. This paper shows how this wide view of trends in the solution space can be used to iteratively search for near optimal co-design solutions. Moreover, our empirical results show that ASCENT produces co-design configurations that average 95%+ optimal for problems with more than 7 points of variability in each of the hardware and software design spaces.

Paper organization. The remainder of this paper is organized as follows: Section 2 presents a motivating example of a satellite hardware/software co-design problem; Section 3 discusses the challenges of solving software/-hardware co-design problems in the context of this motivating example; Section 4 describes the ASCENT heuristic algorithm; Section 5 analyzes empirical results from experiments we performed with ASCENT; Section 6 compares ASCENT with related work; and Section 7 presents concluding remarks and lessons learned from our work with ASCENT.

2 Motivating Example

This section presents a satellite design example to motivate the need to expand search-based software engineering techniques to encompass cost-constrained hardware/software producer/-consumer co-design problems. Designing satellites, such as the satellite for NASA's Magnetospheric Multiscale (MMS) mission [12], requires carefully balancing hardware/software design subject to tight budgets. Figure 1 shows a satellite with a number of possible variations in software and hardware design. For example, the software design has a point of

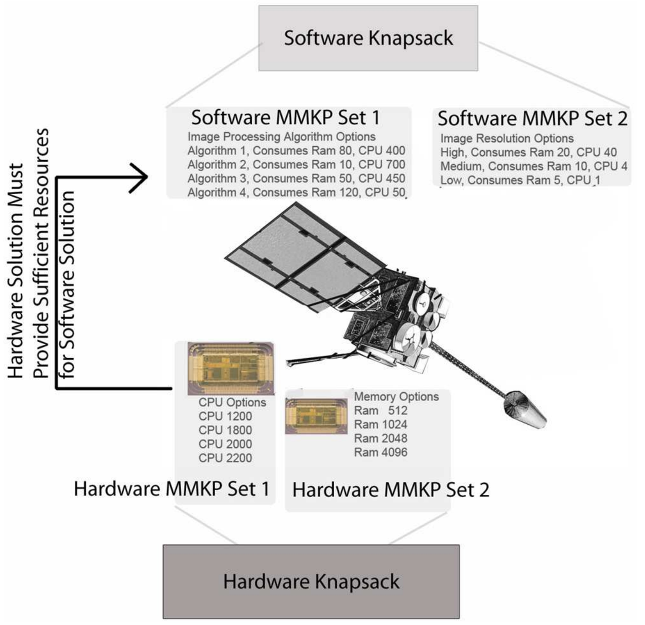

Software Knapsack
Software MMKP Set 1
Image Processing Algorithm Options
Algorithm 1, Consumes Ram 80, CPU 400
Algorithm 2, Consumes Ram 10, CPU 700
Algorithm 3, Consumes Ram 50, CPU 450
Algorithm 4, Consumes Ram 120, CPU 50
Software MMKP Set 2
Image Resolution Options
High, Consumes Ram 20, CPU 40
Medium, Consumes Ram 10, CPU 4
Low, Consumes Ram 5, CPU 1
Hardware Solution Must Provide Sufficient Resources for Software Solution
CPU Options
CPU 1200
CPU 1800
CPU 2000
CPU 2200
Memory Options
Ram 512
Ram 1024
Ram 2048
Ram 4096
Hardware MMKP Set 1
Hardware MMKP Set 2
Hardware Knapsack

Fig. 1. Software/Hardware Design Variability in a Satellite

variability where a designer can select the resolution of the images that are processed. Processing higher resolution images improves the accuracy but requires more RAM and CPU cycles.

Another point of variability in the software design is the image processing algorithms that can be used to identify characteristics of the images captured by the satellite's cameras. The algorithms each provide a distinct level of accuracy, while also consuming different quantities of RAM and CPU cycles. The underlying hardware has a number of points of variability that can be used to increase or decrease the RAM and CPU power to support the resource demands of different image processing configurations. Each configuration option, such as the chosen algorithm or RAM value, has a cost associated with it that subtracts from the overall budget. A key question design question for the satellite is: *what set of hardware and software choices will fit a given budget and maximize the image processing accuracy*.

Many similar design problems involving the allocation of resources subject to a series of design constraints have been modeled as *Multidimensional Multiple-Choice Knapsack Problems* (MMKPs) [20, 22, 2]. A standard knapsack problem [21] is defined by a set of items with varying sizes and values. The goal is to find the set of items that fits into a fixed sized knapsack and that simultaneously maximizes the value of the items in the knapsack. An MMKP problem is a variation on a standard knapsack problem where the items are divided into sets and at most one item from each set may be placed into the knapsack.

Figure 2 shows an example MMKP problem where two sets contain items of different sizes and values. At most

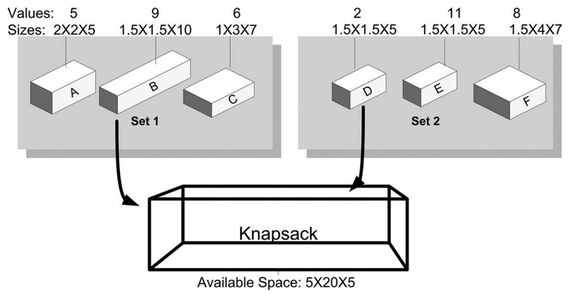


Fig. 2. An Example MMKP Problem

one of the items A,B, and C can be put into the knapsack. Likewise, only one of the items D, E, and F can be put into the knapsack. The goal is to find the combination of two items, where one item is chosen from each set, that fits into the knapsack and maximizes the overall value. A number of resource related problems have been modeled as MMKP problems where the sets are the points of variability in the design, the items are the options for each point of variability, and the knapsack/item sizes are the resources consumed by different design options [25, 22, 33, 8, 3].

The software and hardware design problems are hard to solve individually. Each design problem consists of a number of design variability points that can be implemented by exactly one design option, such as a specific image processing algorithm. Each design option has an associated resource consumption, such as cost, and value associated with it. Moreover, the design options cannot be arbitrarily chosen because there is a limited amount of each resource available to consume.

It is apparent that the description of the software design problem directly parallels the definition of an MMKP problem. An MMKP set can be created for each point of variability (*e*.g., Image Resolution and Algorithm). Each set can then be populated with the options for its corresponding point of variability (*e*.g., High, Medium, Low for Image Resolution). The items each have a size (cost) associated with them and there is a limited size knapsack (budget) that the items can fit into. Clearly, just selecting the optimal set of software features subject to a maximum budget is an instance of the NP-Hard [11] MMKP problem.

For the overall satellite design problem, we must contend with not one but two individual knapsack problems. One problem models the software design and the second problem models the hardware design. The first of the two MMKP problems for the satellite design is its software MMKP problem. The hardware design options are modeled in a separate MMKP problem with each set containing the potential hardware options. An example mapping of the software and hardware design problems to MMKP problems is shown in Figure 1.

We call this combined two problem MMKP model a *MMKP co-design problem*. With this MMKP co-design model of the satellite, a design is reached by choosing one item from each set (*e*.g., an Image Resolution, Algorithm, RAM value, and CPU) for each problem. The correctness of the design can be validated by ensuring that exactly one item is chosen from each set and that the items fit into their respective software and hardware knapsacks. This definition, however, is still not sufficient to model the cost-constrained hardware/-software producer/consumer co-design problem since we have not accounted for the constraint on the total size of the two knapsacks or the production and consumption of resources by hardware and software.

A correct solution must also uphold the constraint that the items chosen for the solution to the software MMKP problem do not consume more resources, such as RAM, than are produced by the items selected for the solution to the hardware MMKP problem. Moreover, the cost of the entire selection of items must be less than the total development budget. We know that solving the individual MMKP problems for the optimal hardware and software design is NP-Hard but we must also determine how hard solving the combined co-design problem is.

In this simple satellite example, there are 192 possible satellite configurations to consider, allowing for exhaustive search to be used. For real industrial scale examples, however, there are a significantly larger number of possibilities which makes it infeasible to use an exhaustive search technique. For example, a system with design choices that can be modeled using 64 MMKP sets, each with 2 items, will have 2^{64} possible configurations. For systems of this scale, manual solving methods are clearly not feasible, which motivates the need for a search-based software engineering technique.

2.1 MMKP Co-design Complexity

Below, we show that MMKP co-design problems are NP-Hard and in need of a search-based software engineering technique. We are not aware of any approximation techniques for solving MMKP co-design problems in polynomial time. This lack of approximation algorithms—coupled with the poor scalability of exact solving techniques—hinders DRE system designers's abilities to optimize software and hardware in tandem.

To show that MMKP co-design problems are NP-Hard, we must build a formal definition of them. We can define an MMKP co-design problem, CoP, as an 8-tuple:

$$CoP =< Pr, Co, S_1, S_2, S, R, Uc(x,k), Up(x,k) >$$

where:

- Pr is the producer MMKP problem (*e*.g., the hardware choices).
- Co is the consumer MMKP problem (*e*.g., the software choices).
- S_1 is the size of the producer, Pr, knapsack.
- S_2 is the size of the consumer, Co, knapsack.
- R is the set of resource types (*e*.g., RAM, CPU, etc.) that can be produced and consumed by Pr and Co, respectively.
- S is the total allowed combined size of the two knapsacks for Pr and Co (*e*.g., total budget).
- $Uc(x,k)$ is a function which calculates the amount of the resource $k \subset R$ **c**onsumed by an item $x \subset Co$ (*e*.g., RAM consumed).
- $Up(x,k)$ is a function which calculates the amount of the the resource $k \subset R$ **p**roduced by an item $x \subset Pr$ (*e*.g., RAM provided).

Let a solution to the MMKP co-design problem be defined as a 2-tuple, $< p, c >$, where $p \subset Pr$ is the set of items chosen from the producer MMKP problem and $c \subset Co$ is the set of items chosen from the consumer MMKP problem. A visualization of a solution tuple is shown in Figure 3. We

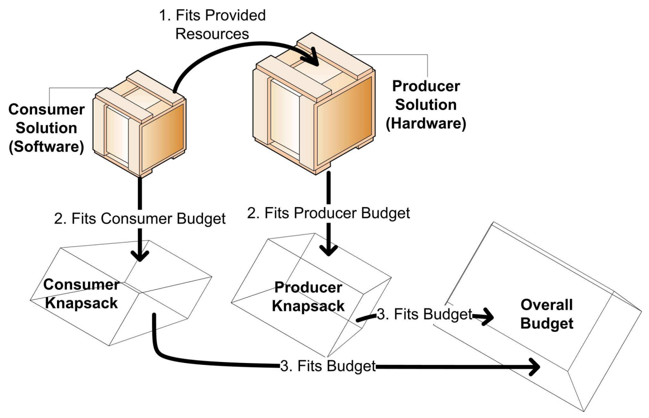


Fig. 3. Structure of an MMKP Co-design Problem

define the value of the solution as the sum of the values of the elements in the consumer solution:

$$V = \sum_{0}^{j} valueof(c_j)$$

where j is the total number of items in c, c_j is the j_{th} item in c, and $valueof()$ is a function that returns the value of an item in the consumer soution.

We require that p and c are valid solutions to Pr and Co, respectively. For p and c to be valid, exactly one item from each set in Pr and Co must have been chosen. Moreover, the items must fit into the knapsacks for Pr and Co. [1]This constraint corresponds to Rule (2) in Figure 3 that each solution must fit into the budget for its respective knapsack.

1. A more formal definition of MMKP solution correctness is available from [2].

The MMKP co-design problem adds two additional constraints on the solutions p and c. First, we require that the items in c do not consume more of any resource than is produced by the items in p:

$$(\forall k \subset R), \sum_{0}^{j} Uc(c_j, k) \leq \sum_{0}^{l} Up(p_l, k)$$

where j is the total number of items in c, c_j is the j_{th} item in c, l is the total number of items in p, and p_j is the j_{th} item in p. Visually, this means that the consumer solution can fit into the producer solution's resources as shown in Rule (1) in Figure 3.

The second constraint on c and p is an interesting twist on traditional MMKP problems. For a MMKP co-design problem, we do not know the exact sizes, S_1, S_2, of each knapsack. Part of the problem is determining the sizes as well as the items for each knapsack. Since we are bound by a total overall budget, we must ensure that the sizes of the knapsacks do not exceed this budget:

$$S_1 + S_2 \leq S$$

This constraint on the overall budget corresponds to Rule (3) in Figure 3.

To show that solving for an exact answer to the MMKP problem is NP-Hard, we will show that we can reduce any instance of the NP-complete *knapsack decision problem* to an instance of the MMKP co-design problem. The knapsack decision problem asks if there is a combination of items with value at least V that can fit into the knapsack without exceeding a cost constraint.

A knapsack problem can easily be converted to a MMKP problem as described by Akbar et al. [2]. For each item, a set is created containing the item and the $\emptyset$ item. The $\emptyset$ item has no value and does not take up any space. Using this approach, a knapsack decision problem, K_{dp}, can be converted to a MMKP decision problem, M_{dp}, where we ask if there is a selection of items from the sets that has value at least V.

To reduce the decision problem to an MMKP co-design problem, we can use the MMKP decision problem as the consumer knapsack (Co = M_{dp}), set the producer knapsack to an MMKP problem with a single item with zero weight and value ($\emptyset$), and let our set of produced and consumed resources, R, be empty, $R = \emptyset$. Next, we can let the total knapsack size budget be the size of the decision problem's knapsack, $S = sizeof(M_{dp})$.

The co-design solution, which is the maximization of the consumer knapsack solution value, will also be the optimal answer for the decision problem, M_{dp}. We have thus setup the co-design problem so that it is solving for a maximal answer to M_{dp} without any additional producer/consumer constraints or knapsack size considerations. Since any instance of the NP-complete knapsack decision problem can be reduced to an MMKP co-design problem, the MMKP co-design problem must be NP-Hard.

3 Challenges of MMKP Co-design Problems

This section describes the two main challenges to building an approximation algorithm to solve MMKP co-design problems. We discuss the challenges that make it infeasible to directly apply existing MMKP algorithms to MMKP co-design problems. The first challenge is that determining how to set the budget allocations of the software and hardware is not straightforward since it involves figuring out the precise size of the software and hardware knapsacks where the hardware knapsack produces sufficient resources to support the optimal software knapsack solution (which itself is unknown). The second challenge is that the tight-coupling between producer and consumer MMKP problems makes them hard to solve individually, thus motivating the need for a heuristic to decouple them.

3.1 Challenge 1: Undefined Producer/Consumer Knapsack Sizes

One challenge of the MMKP co-design problem is that the individual knapsack size budget for each of the MMKP problems is not predetermined, *i.e.*, we do not know how much of the budget should be allocated to software versus hardware, as shown in Figure 4. The only constraint is that the sum of the

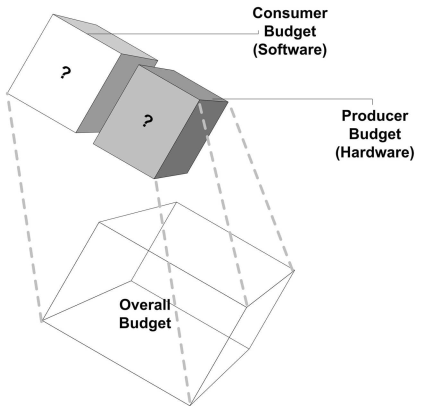


Fig. 4. Undefined Knapsack Sizes

budgets must be less than or equal to an overall total budget. Every pair of budget values for hardware and software results in two new unique MMKP problems. Even minor transfers of capital from one problem budget to the other can therefore completely alter the solution of the problem, resulting in a new maximum value. Existing MMKP techniques assume that the exact desired size of the knapsack is known.

There is currently no information to aid designers in determining the allocation of the budgets. As a result, many designers may choose the allocation arbitrarily without realizing the profound impact it may have. For example, a budget allocation of 75% software and 25% software may result in a solution that, while valid, provides far less value and costs considerably more than a solution with a budget allocation of 74% and 26% percent.

There is, however, useful information in the solution space that can be determined by solving instances of the problem with unique sequential divisions of the total budget. Typically, designers would choose a budget with little information on the ramifications of a potential budget choice. By sampling the solution space at a number of distinct budget allocations, the algorithm can show designers the best solution that it can produce at each budget allocation. The information produced, may not show the actual best budget allocations to choose, but should help designers to make better budget allocation choices than blindly choosing a budget allocation with no information at all. A key challenge is figuring out how to sample the solution space and present the results to designers. In Section 4.4 we discuss ASCENT's solution to this problem and in Section 5 we present empirical data showing how ASCENT allows designers to sample design spaces for a number of MMKP co-design problems.

3.2 Challenge 2: Tight-coupling Between the Producer/Consumer

Another key issue to contend with is how to rank the solutions to the producer MMKP problem. Per the definition of an MMKP co-design problem from Section 2.1, the producer solution does not directly impart any value to the overall solution. The producer's benefit to a solution is its ability to make a good consumer solution viable. MMKP solvers must have a way of ranking solutions and items. The problem, however, is that the value of a producer solution or item cannot be calculated in isolation.

A consumer solution must already exist to calculate the value of a particular producer solution. For example, whether or not 1,024 kilobytes of memory are beneficial to the overall solution can only be ascertained by seeing if 1,024 kilobytes of memory are needed by the consumer solution. If the consumer solution does not need this much memory, then the memory produced by the item is not helpful. If the consumer solution is RAM starved, the item is desperately needed. A visualization of the problem is shown in Figure 5.

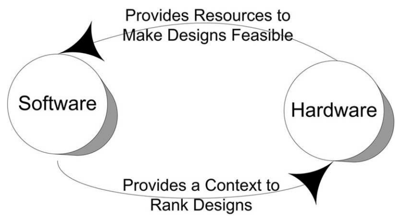

Provides Resources to Make Designs Feasible
Software
Hardware
Provides a Context to Rank Designs

Fig. 5. Producer/Consumer MMKP Tight-coupling

The inability to rank producer solutions in isolation of consumer solutions is problematic because it creates a chicken and the egg problem. A valid consumer solution cannot be chosen if we do not know what resources are available for it to consume. At the same time, we cannot rank the value of producer solutions without a consumer solution as a context. This tight-coupling between the producer/consumer is a challenging problem. We discuss the heuristic ASCENT uses to solve this problem in Section 4.3.

4 THE ASCENT ALGORITHM

This section presents our polynomial-time approximation algorithm, called the *Allocation-baSed Configuration ExploratioN Technique* (ASCENT), for solving MMKP co-design problems. The pseudo-code for the ASCENT algorithm is shown in Figure 6 and explained throughout this section.

4.1 ASCENT Algorithm Overview

A MMKP co-design problem, CoP, as defined as an 8-tuple:

$$CoP =< Pr, Co, S_1, S_2, S, R, Uc(x,k), Up(x,k) >$$

The ASCENT algorithm solves for a series of potential solutions to CoP using an iterative heuristic algorithm. The input to the algorithm is the problem definition CoP and a step size increment, D, which is discussed in Section 4.2. The ASCENT algorithm then proceeds as shown in Figure 6

4.2 Producer/Consumer Knapsack Sizing

The first issue to contend with when solving an MMKP co-design problem is Challenge 2 from Section 3.1, which involves determining how to allocate sizes to the individual knapsacks. ASCENT addresses this problem by dividing the overall knapsack size budget into increments of size D. The size increment is a parameter provided by the user. ASCENT then iteratively increases the consumer's budget allocation (knapsack size) from 0% of the total budget to 100% of the total budget in steps of size D. The incremental expansion of the producer's budget can be seen in the `for` loop in step 1 of Figure 6 and the setting of values for of S_1, S_2 in step 2.

For example, if there is a total size budget of 100 and increments of size 10, ASCENT firsts assign 0 to the consumer and 100 to the producer, 10 and 90, 80 and 20, and so forth until 100% of the budget is assigned to the consumer. Since we are focused on DRE systems, we assume that the resources of the system are fixed and that 100% utilization cannot be exceeded. The allocation process is shown in Figure 7. ASCENT includes both the 0%,100% and 100%,0% budget allocations to handle cases where the optimal configuration includes producer or consumer items with zero cost.

4.3 Ranking Producer Solutions

At each allocation iteration, ASCENT has a fixed set of sizes for the two knapsacks. In each iteration, ASCENT must solve the coupling problem presented in Section 3.2, which is: how do we rank producer solutions without a consumer solution. After the coupling is loosened, ASCENT can solve for a highly valued solution that fits the given knapsack size restrictions.

To break the tight-coupling between producer and consumer ordering, ASCENT employs a special heuristic. Once the knapsack size allocations are fixed, ASCENT solves for a maximal consumer solution that only considers the current size constraint of its knapsack and not produced/consumed resources. This process is shown in step 2.1 of Figure 6.

The process of solving the consumer knapsack Co, in Step 2.1, uses an arbitrary MMKP approximation algorithm to find

```
Inputs:

    $CoP = < Pr, Co, S_1, S_2, S, R, Uc(x,k), Up(x,k) >$
    $D = stepsize$

Algorithm:
  1) For $int\ i = 0$ to $\lfloor S/D \rfloor$, set $S_1 = i * D$ and $S_2 = S - S_1$
  2) For each set of values for $S_1$ and $S_2$:
     2.1) Solve for a solution, $tc$, to $Co$, given $S_2$
     2.2) Calculate a resource consumption heuristic $Vr(k)$ for the resource in $r \in R$:
          $Vr(r) = \frac{\sum_{j=0}^{|tc|} Uc(tc_j, k)}{\sum_{j=0}^{|R|} \sum_{k=0}^{|tc|} Uc(tc_j, k)}$
     2.3) Solve for a solution, $p$, to $Pr$ that maximizes the sum of the values of the items selected for the knapsack, $\sum_{k=0}^{|p|} Value(p_k)$, where the value of the $k_{th}$ item is calculated as:
          $Value(p_k) = \sum_{j=0}^{|R|} Vr(r_j) * Up(p_k, r_j)$
     2.4) For each resources $r_j \in R$, calculate the amount of that resource, $P(r)$, produced by the items in $p$:
          $P(r) = Up(p_0, r_j) + Up(p_1, r_j) \ldots Up(p_{|p|-1}, r_j)$
     2.5) Create a new multidimensional knapsack problem, $Cmo$, from $Co$, such that the maximum size of each dimension of the new knapsack is defined by the vector:
          $Sm_2 = (S_2, r_0, r_1, \ldots r_{|R|-1})$
     2.6) Solve for a solution, $c$, to $Cmo$ and add a solution tuple $< p, c >$ to the list of candidate solutions, $lc$, for $CoP$
  3) Sort the potential solutions, $lc$, of $CoP$ and output both the highest valued solution and the list of other potential solutions
```

Fig. 6. The ASCENT Algorithm

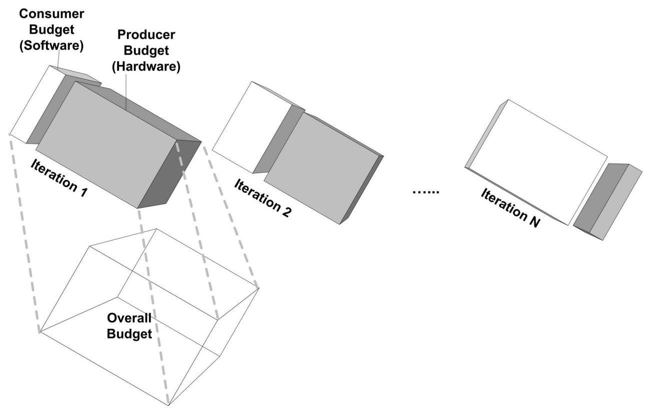


Fig. 7. Iteratively Allocating Budget to the Consumer Knapsack

a solution that only considers the consumer's budget. This approach is similar to asking "what would the best possible solution look like if there were unlimited produced/consumed resources." Once ASCENT has this idealized consumer solution, it calculates a heuristic for assigning a value to producer solutions.

ASCENT uses a commonly used heuristic from prior work on MMKPs to assign values for ranking potential solutions [2]. The heuristic that ASCENT uses to assign value to producer items is: *how valuable are the resources of a producer item to the idealized consumer solution*. This heuristic is calculated as a set of values for the V_r variables in Step 2.2 of Figure 6. We calculate the value of a resource as the amount of the resource consumed by the idealized consumer solution divided by the sum of the total resources consumed by the solution. In Step 2.3 of Figure 6, the resource ratios (V_r values) are known and each item in the producer MMKP problem is assigned a value by multiplying each of its provided resource values by the corresponding ratio and summing the results.

4.4 Solving the Individual MMKP Problems

Once sizes have been set for each knapsack and the valuation heuristic has been applied to the producer MMKP problem, existing MMKP solving approaches can be applied. First, the producer MMKP problem, with its new item values, is solved for an optimal solution, as shown in Step 2.3 of Figure 6. In Step 2.5, a new consumer MMKP problem is created with constraints reflecting the maximum available amount of each resource produced by the solution from the producer MMKP problem. The consumer MMKP problem is then solved for an solution in Step 2.6. The producer and consumer solutions are then combined into the 2-tuple, $< p, c >$ and saved.

In each iteration, ASCENT assigns sizes to the producer and consumer knapsacks and the solving process is repeated. A collection of the 2-tuple solutions is compiled during the process. The output of ASCENT, returned in Step 3 of Figure 6, is both the 2-tuple with the greatest value and the collection of 2-tuples.

The reason that the 2-tuples are saved and returned as part of the output is that they provide valuable information on the trends in the solution space of the co-design problem. Each 2-tuple contains a high-valued solution to the co-design problem at a particular ratio of knapsack sizes. This data can be used to graph and visualize how the overall solution value changes as a function of the ratio of knapsack sizes. As shown in Section 5, this information can be used to ascertain a number of useful solution space characteristics, such as determining how much it costs to increase the value of a specific system property to a given level or finding the design with the highest value per unit of cost.

4.5 Algorithmic Complexity

The overall algorithmic complexity of ASCENT can be broken down as follows:

1) there are T iterations of ASCENT
2) in each iteration there are 3 invocations to an MMKP approximation algorithm
3) in each iteration, values of at most n producer items must be updated.

This breakdown yields an algorithmic complexity of O($T(n+MMKP)$), where MMKP is the algorithmic complexity of the chosen MMKP algorithm. With M-HEU (one of the most accurate MMKP approximation algorithms [2]) the algorithmic complexity is O($mn^2(l-1)^2$), where m is the number of resource types, n is the number of sets, and l is maximum items per set. Our experiments in Section 5 used $T = 100$ and found that it provided excellent results. With our experimental setup that used M-HEU, the overall algorithmic complexity was therefore O($100(mn^2(l-1)^2+n)$). This algorithmic complexity is polynomial and thus ASCENT should be able to scale up to very large problems, such as the co-design of production satellite hardware and software.

5 Analysis of Empirical Results

This section presents empirical data we obtained from experiments using ASCENT to solve MMKP co-design problems. The empirical results demonstrate that ASCENT produces solutions that are often near the maximum value that can be achieved while not exceeding resource constraints. The results also show that ASCENT can not only provide near optimal designs for the co-design problems, such as the satellite example, but also scale to the large problem sizes of production DRE systems. Moreover, we show that the data sets generated by ASCENT—which contain high valued solutions at each budget allocation—can be used to perform a number of important search-based software engineering studies on the co-design solution space.

Our empirical results also compare the performance of ASCENT to implementations of Genetic and Particle Swarm Optimization (PSO) algorithms for solving the MMKP Co-design problem. The results show that ASCENT produces solutions with superior optimality. At the same time, ASCENT runs roughly 10 times faster than either the Genetic or PSO algorithms.

Each experiment used a total of 100 budget iterations ($T = 100$). We also used the M-HEU MMKP approximation algorithm as our MMKP solver. All experiments were conducted on an Apple MacBook Pro with a 2.4 GHz Intel Core 2 Duo processor, 2 gigabyes of RAM, running OS X version 10.4.11, and a 1.5 Java Virtual Machine (JVM) run in client mode. The JVM was launched with a maximum heap size of 64mb (-Xmx=64m).

We chose the M-HEU algorithm since it is straight-forward to implement and provided good results in our initial experiments. Many other excellent MMKP heuristic algorithms are available that may produce better results at the expense of increased solving time and implementation complexity. ASCENT does not require the use of any specific MMKP algorithm, such as M-HEU, and thus designers can choose alternate MMKP heuristic algorithms if they prefer.

5.1 MMKP Co-design Problem Generation

A key capability needed for the experiments was the ability to randomly generate MMKP co-design problems for test data.

For each problem, we also needed to calculate how good ASCENT's solution was as a percentage of the optimal solution: $\frac{valueof(ASCENTSolution)}{valueof(OptimalSolution)}$. For small problems with less than 7 sets per MMKP problem, we were able to use a constraint logic programming (CLP) [31] technique built on top of the Java Choco constraint solver (`choco-solver.net`) to derive the optimal solution.

For larger scale problems the CLP technique was simply not feasible, *e*.g., solutions might take years to find. For larger problems, we developed a technique that randomly generated MMKP co-design problems with a few carefully crafted constraints so we knew the exact optimal answer. Others [2] have used this general approach, though with a different problem generation technique.

Ideally, we would prefer to generate completely random problems to test ASCENT. We are confident in the validity of this technique, however, for two reasons: (1) the trends we observed from smaller problems with truly random data were identical to those we saw in the data obtained from solving the generated problems and (2) the generated problems randomly placed the optimal items and randomly assigned their value and size so that the problems did not have a structure clearly amenable to the heuristics used by our MMKP approximation algorithm. We did not use Akbar's technique [2] because the problems it generated were susceptible to a greedy strategy.

Our problem generation technique worked by creating two MMKP problems for which we knew the exact optimal answer. First, we will discuss how we generated the individual MMKP problems. Let S be the set of MMKP sets for the problem, $\vec{R}$ be a K-dimensional vector describing the size of the knapsack, I_{ij} be the j_{th} item of the i_{th} set, $size(I_{ij}, k)$ be the k_{th} component of I_{ij}'s size vector $\vec{Sz_{ij}}$, and $size(S, k)$ be the k_{th} component of the knapsack size vector, the problem generation technique for each MMKP problem worked as follows:

1) Randomly populate each set, $s \subset S$, with a number of items
2) Generate a random size, $\vec{R}$, for the knapsack
3) Randomly choose one item, $Iopt_i \subset OptItems$ from each set to be the optimal item. $Iopt_i$ is the optimal item in the i_{th} set.
4) Set the sizes of the items in $OptItems$, so that when added together they exactly consume all of the space in the knapsack:
$$(\forall k \subset R), (\sum_0^i size(Iopt_i, k)) = size(S, k)$$
5) Randomly generate a value, $Vopt_i$, for the optimal item, $Iopt_i$, in each set
6) Randomly generate a value delta variable, $V_d < min(Vopt_i)$, where $min(Vopt_i)$ is the optimal item with the smallest value
7) Randomly set the size and values of the remaining non-optimal items in the sets so that either:
   - The item has a greater value than the optimal item in its set. In this case, each component of the item's size vector, is greater than the corresponding component in the optimal item's size vector: $(\forall k \subset R), size(Iopt_i, k) < size(I_{ij}, k)$
   - The item has a smaller value than the optimal item's value minus V_d, $valueof(I_{ij}) < Vopt_i - V_d$. This constraint will be important in the next step. In this case, each component of the item's size vector is randomly generated.

At this point, we have a very random MMKP problem. What we have to do is further constrain the problem so that we can guarantee the items in $OptItems$ are truly the optimal selection of items. Let $MaxV_i$ be the item with the highest value in the i_{th} set. We further constrain the problem as follows:

For each item $MaxV_i$, we reset the values of the items (if needed) to ensure that the sum of the differences between the max valued items in each set and the optimal item are less than V_d:

$$\sum_0^i (MaxV_i - Vopt_i) < V_d$$

A visualization of this constraint is shown in Figure 8.

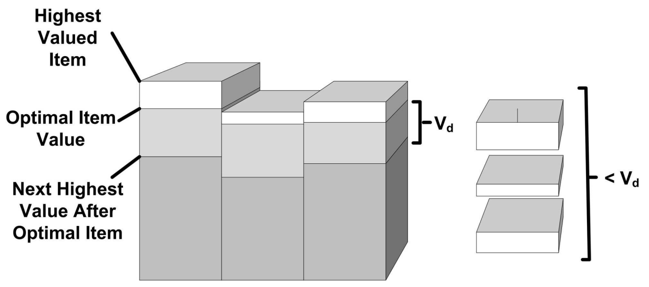


Fig. 8. A Visualization of V_d

This new valuation of the items guarantees that the items in $OptItems$ are the optimal items. We can prove this property by showing that if it does not hold, there is a contradiction. Assume that there is some set of items, $Ibetter$, that fit into the knapsack and have a higher value. Let Vb_i be the value of the better item to choose than the optimal item in the i_{th} set. The sum of the values of the better items from each set must have a higher value than the optimal items.

The items $Ib_i \subset Ibetter$ must fit into the knapsack. We designed the problem so that the optimal items exactly fit into the knapsack and that any item with a higher value than an optimal item is also bigger. This design implies that at least one of the items in $Ibetter$ is smaller and thus also has a smaller value, $Vsmall$, than the optimal item in its set (or $Ibetter$ wouldn't fit). If there are Q sets in the MMKP problem, this implies that at most $Q-1$ items in $Ibetter$ have a larger value than the optimal item in their set, and thus:

$$Vopt_Q + \sum_0^{Q-1} Vopt_i < Vsmall + \sum_0^{Q-1} Vb_i$$

We explicitly revalued the items so that:

$$\sum_0^i (MaxV_i - Vopt_i) < V_d$$

By subtracting the $\sum_0^{Q-1} Vopt_i$ from both sides, we get:

$$Vopt_Q < Vsmall + \sum_0^{Q-1}(Vb_i - Vopt_i)$$

the inequality will still hold if we substitute V_d in for $\sum_0^{Q-1}(Vb_i - Vopt_i)$, because V_d is larger:

$$Vopt_Q < Vsmall + V_d$$

$$Vopt_Q - V_d < Vsmall$$

which is a contradicton of the rule that we enforced for smaller items: $valueof(I_{ij}) < Vopt_i - V_d$

This problem generation technique creates MMKP problems with some important properties. First, the optimal item in each set will have a random number of larger and smaller valued items (or none) in its set. This property guarantees that a greedy strategy will not necessarily do well on the problems.

Moreover, the optimal item may not have the best ratio of value/size. For example, an item valued slightly smaller than the optimal item may consume significantly less space because its size was randomly generated. Many MMKP approximation algorithms use the value/size heuristic to choose items. Since there is no guarantee on how good the value/size of the optimal item is, MMKP approximation algorithms will not automatically do well on these problems.

To create an MMKP co-design problem where we know the optimal answer, we generate a single MMKP problem with a known optimal answer and split it into two MMKP problems to create the producer and consumer MMKP problems. To split the problem, two new MMKP problems are created. One MMKP problem receives E of the sets from the original problem and the other problem receives the remaining sets. The total knapsack size for each problem is set to exactly the size required by the optimal items from its sets to fit. The sum of the two knapsack sizes will equal the original knapsack size. Since the overall knapsack size budget does not change, the original optimal items remain the overall optimal solution.

Next, we generate a set of produced/consumed resource values for the two MMKP problems. For the consumer problem, we randomly assign each item an amount of each produced resource $k \subset R$ that the item consumes. Let $TotalC(k)$ be the total amount of the resource k needed by the optimal consumer solution and $Vopt(p)$ be the optimal value for the producer MMKP problem. We take the consumer problem and calculate a resource production ratio, $Rp(k)$, where

$$Rp(k) = \frac{TotalC(k)}{Vopt(p)}$$

For each item, I_{ij}, in the producer problem, we assign it a production value for the resource k of: $Produced(k) = Rp(k) * valueof(I_{ij})$.

The optimal items have the highest feasible total value based on the given budget and the sum of their values times the resource production ratios exactly equals the needed value of each resource k:

$$TotalC(k) = \frac{TotalC(k)}{Vopt(p)} * \sum_0^{i} Vopt_i$$

Any other set of items must have a smaller total value and consequently not provide sufficient resources for the optimal set of consumer items. To complete the co-design problem, we set the total knapsack size budget to the sum of the sizes of the two individual knapsacks.

5.2 Comparison of ASCENT, a Genetic Algorithm, and PSO

Experiment 1: Comparing ASCENT's Optimality Versus a Genetic Algorithm, and PSO. For our first experiment, we created semi-random MMKP co-design problems that we knew the optimal answer to using the technique from Section 5.1. We generated MMKP co-design problems that ranged in size from 2 to 30 sets per MMKP. Each set contained 15 items. These experiments yielded solution space sizes of between 15^4 and 15^{60} (2 problems with 30 sets of 15 items). For each problem size, we generated and solved 30 problem instances using ASCENT, a genetic algorithm, and a PSO algorithm. We graphed and compared the optimality and solving time of the three algorithms.

Genetic/PSO Design: The Genetic and PSO algorithms both used a common representation of the problem and penalty function. The problem was represented as an n-dimensional vector, where the positions in the vector corresponded to the item that was selected from each set. For example, a problem with 3 sets per MMKP problem would produce a vector with 6 components. The first 3 components would represent the items selected from the consumer MMKP problem's sets. The second 3 components would represent the items selected from the producer MMKP problem's sets.

Each position in the vector was limited to values corresponding to the valid 0-based indices of items in the sets. For example, a set with 5 items would allow values of 0-4. A value of 2 would correspond to selecting the 3^{rd} item in the set.

The penalty function scored solutions based on 1) if the solution's overall value and 2) whether or not the solution was correct. If a solution was not valid, the score of the solution was set to 0 - (resource overconsumption). That is, solutions that did not properly adhere to the budget or the production and consumption of resources would produce negative values. Although repair functions can sometimes provide better results than a penalizing function, repairing an arbitrary invalid MMKP co-design solution is extremely complex and a research endeavor in its own right.

For the genetic and PSO algorithms, we used population (total particle) sizes of 20, 200, and 2000 members. We conducted various experiments to tune the parameters of the algorithms. For the PSO algorithm, we used a local learning rate of 2, a global learning rate of 2, and an inertial value of 0.5. For the Genetic algorithm, we mated the top 25% of the population using a strict cutoff. We also allowed up to 50% of the population to survive and cross over from one generation to the next. Finally, we used a mutation probability of 0.05%. Each algorithm was run for a total of 20 generations/iterations.

Experiment Results with Semi-Random Data: The results for the first experiment are shown in Figure 9.

As can be seen from the results, for up to 5 sets per MMKP problem, the Genetic algorithm with 2000 population

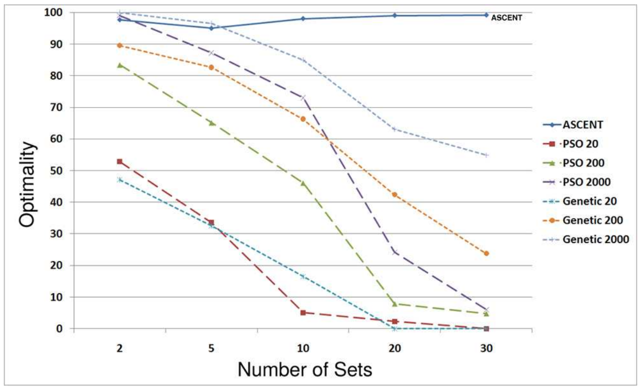


Fig. 9. Solution Optimality vs Number of Sets Compared for ASCENT, a Genetic Algorithm, and a PSO Algorithm

members provided the best results. The Genetic Algorithm with 2000 population members, required 9,024ms to solve a problem with 5 sets. ASCENT, in contrast, required 73ms. When the problems were scaled up to 30 sets per MMKP problem, ASCENT provided far superior optimality and run time. ASCENT produced solutions that averaged roughly 99.2% optimal versus the Genetic algorithms 54.9% optimality. Furthermore, ASCENT solved the problems in an average of 317ms versus the Genetic algorithms average runtime of 64,212ms.

Experiment Results with Random Data: We also compared the algorithms on a series of problems that were completely randomly generated. For these problems, we did not know the true optimal value. We generated 100 problems with 50 sets per MMKP problem and 15 items per set. This yielded a solution space size of 15^{100}. In order to ensure that we generated tractable problem instances, we set extremely loose resource constraints on the problems to create a high probability that a solution existed.

Figure 10 shows a graph of the solution scores of the algorithms on these 100 random problems.

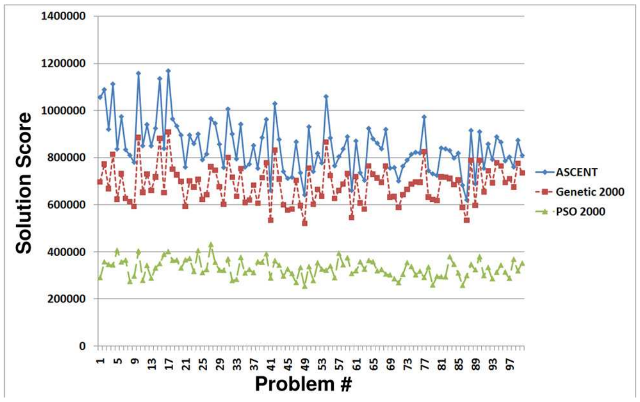


Fig. 10. Solution Score for 100 Randomly Generated Problems

As can be seen from Figure 10, ASCENT produced superior solution scores across all 100 problem instances. The Genetic algorithm, which was the second best algorithm, produced solutions that were at most 90.9% of the value of ASCENT's solution and at least 65.8% of the value. The PSO produced solutions that were at most 43.5% of the value of ASCENT's solutions and at least 27.5% of the value. The average solving time for the Genetic 2000 was 101,453ms. The average solving time for the PSO 2000 was 55,203ms. The average solving time for ASCENT was 672ms.

5.3 ASCENT Scalability and Optimality

Experiment 2: Comparing ASCENT scalability to an exact CLP technique. When designing a satellite it is critical that designers can gauge the accuracy of their design techniques. Moreover, designers of a complicated satellite system need to know how different design techniques scale and which technique to use for a given problem size. This set of experiments evalutes these questions for ASCENT and a constraint logic programming (CLP) co-design technique.

Although CLP solvers can find optimal solutions to MMKP co-design problems they have exponential time complexity. For large-scale co-design problems (such as designing a complicated climate monitoring satellite) CLP solvers thus quickly become incapable of finding a solution in a reasonable time frame. We setup an experiment to compare the scalability of ASCENT to an CLP technique. We randomly generated a series of problems ranging in size from 1 to 7 sets per hardware and software MMKP problem. Each set had 10 items. We tracked and compared the solving time for ASCENT and the CLP technique as the number of sets grew. Figure 11 presents the results from the experiment. As shown by the

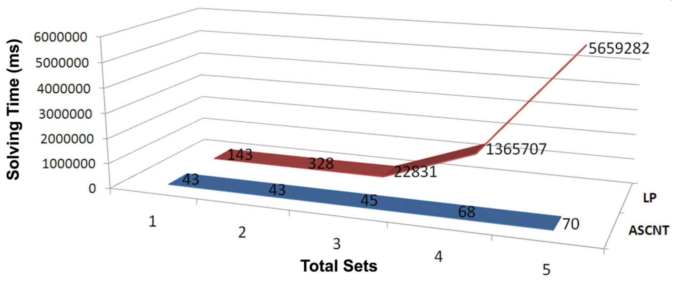


Fig. 11. Solving Time for ASCENT vs. CLP

results, ASCENT scales significantly better than an CLP-based approach.

Experiment 3: Testing ASCENT's solution optimality. Clearly, scalability alone is not the only characteristic of a good approximation algorithm. A good approximation algorithm must also provide very optimal results for large problem sizes. We created an experiment to test the accuracy of ASCENT's solutions. We compared the value of ASCENT's answer to the optimal answer,

$$\frac{valueof(ASCENTSolution)}{valueof(OptimalSolution)}$$

for 50 different MMKP co-design problem sizes with 3 items per set. For each size co-design problem, we solved 50 different problem instances and averaged the results.

It is often suggested, due to the Central Limit Theorem [19], to use a sample size of 30 or larger to produce an approximately normal data distribution [15]. We chose a sample size of 50 to remain well above this recommended minimum sample size. The largest problems, with 50 sets per MMKP problem, would be the equivalent of a satellite with 50 points of software variability and an additional 50 points of hardware variability.

For problems with less than 7 sets per MMKP problem, we compared against the optimal answer produced with an CLP solver. We chose a low number of items per set to decrease the time required by the CLP solver and make the experiment feasible. For problems with more than 7 sets, which could not be solved in a timely manner with the CLP technique, we used our co-design problem generation technique presented in Section 5.1. The problem generation technique allowed us to create random MMKP co-design problems that we knew the exact optimal answer for and could compare against ASCENT's answer.

Figure 12 shows the results of the experiment to test ASCENT's solution value verusus the optimal value over 50 MMKP co-design problem sizes.

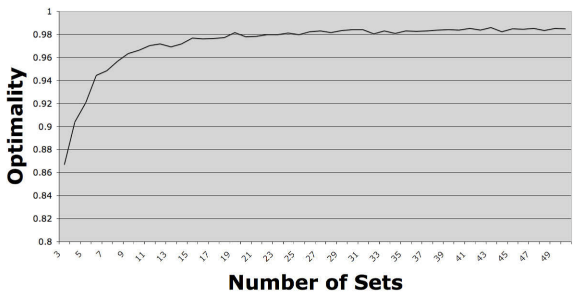


Fig. 12. Solution Optimality vs Number of Sets

With 5 sets, ASCENT produces answers that average 90% optimal. With 7 sets, the answers average ∼95% optimal. Beyond 20 sets, the average optimality is ∼98% and continues to improve. These results are similar to MMKP approximation algorithms, such as M-HEU, that also improve with increasing numbers of sets [2]. We also found that increasing the number of items per set also increased the optimality, which parallels the results for our solver M-HEU [2].

Experiment 4: Measuring ASCENT's solution space snapshot accuracy. As part of the solving process, ASCENT not only returns the optimal valued solution for a co-design problem but it also produces a data set to graph the optimal answer at each budget allocation. For the satellite example, the graph would show designers the design with the highest image processing accuracy for each ratio of budget allocation to software and hardware. We created an experiment to test how optimal each data point in this graph was.

For this experiment, we generated 100 co-design problems with less than 7 sets per MMKP problem and compared ASCENT's answer at each budget allocation to the optimal answer derived using an CLP technique (more sets improves ASCENT's accuracy). For problems with 7 sets divided into 98 different budget allocations, ASCENT finds the same, optimal solution as the CLP solver more than 85% of the time. Figure 13 shows an example that compares the solution space graph produced by ASCENT to a solution space graph produced with an CLP technique. The X-axis shows the

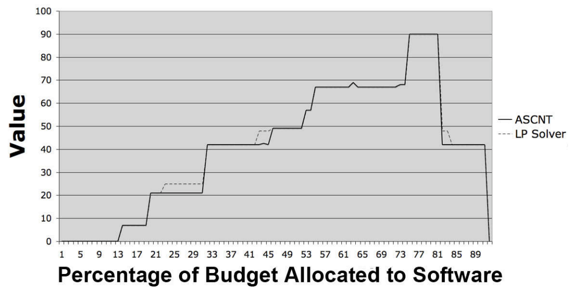


Fig. 13. Solution Value vs. Budget Allocation

percentage of the budget allocated to the software (consumer) MMKP problem. The Y-axis shows the total value of the MMKP co-design problem solution. The ASCENT solution space graph closely matches the actual solution space graph produced with the CLP technique.

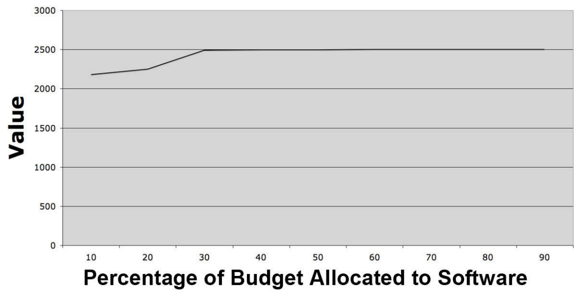
5.4 Solution Space Snapshot Resolution

Experiment 5: Demonstrating the importance of solution space snapshot resolution. A complicated challenge of applying search-based software engineering to hardware/software co-design problems is that design decisions are rarely as straightforward as identifying the design configuration that maximizes a specific property. For example, if one satellite configuration provides 98% of the accuracy of the most optimal configuration for 50% less cost, designers are likely to choose it. If designers have extensive experience in hardware development, they may favor a solution that is marginally more expensive but allocates more of the development to hardware, which they know well. Search-based software engineering techniques should therefore allow designers to iteratively tease these desired designs out of the solution space.

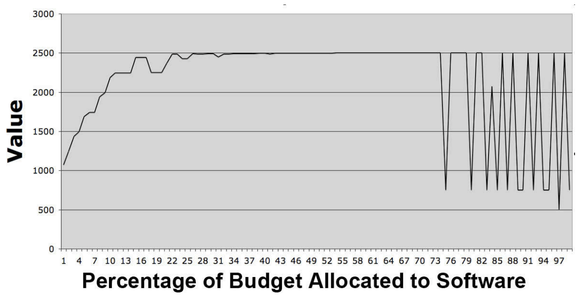
ASCENT has a number of capabilities beyond simply finding the optimal solution for a problem to help designers find desirable solutions. First, as we describe below, ASCENT can be adjusted to produce different resolution images of the solution space by adjusting the granularity of the budget allocation steps (*e*.g., make smaller and more allocation changes). ASCENT's other solution space analysis capabilities are presented in Section 5.5.

The granularity of the step size greatly impacts the resolution or detail that can be seen in the solution space. To obtain the most accurate and informative solution space image, a small step size should be used. Figure 14(a) shows a solution space graph generated through ASCENT using 10 allocation steps. The X-axis is the percentage of budget allocated to software, the Y-axis is the total value of the solution. It appears that any allocation of 30% or more of the budget to software will produce a satellite with optimal image processing accuracy.

The importance of a small step size is demonstrated in Figure 14(b), which was produced with 100 allocation steps. Figure 14(a) suggests that any allocation of greater than 30% for software would result in an optimal satellite design. Figure 14(b) shows that there are many pitfalls in the 70%



(a) Low Resolution Solution Space Snapshot



(b) High Resolution Solution Space Snapshot

Fig. 14. A Solution Space Graph at Varying Resolutions

to 99% range that must be avoided. At these precise budget allocation points, there is not a good combination of hardware and software that will produce a good solution.

5.5 Solution Space Analysis with ASCENT

Although ASCENT's ability to provide variable resolution solution space images is important, its greatest value stems from the variety of questions that can be answered from its output data. In the following results, we present representative solution space analyses that can be performed with ASCENT's output data.

Design analysis 1: Finding designs that produce budget surpluses. Designers may wish to know how the resource slack values, such as how much RAM is unused, with different satellite designs. Another related question is how much of the budget will be left-over for designs that provides a specified minimal level of image processing accuracy. We can use the same ASCENT output data to graph the budget surplus at a range of allocation values.

Figure 15 shows the budget surplus from choosing various designs. The graph has been filtered to adhere to a requirement that the solution provide a value of at least 1600. Any data point with a value of less than 1600 has had its surplus set to 0. Looking at the graph, we can see that the cheapest design that provides a value of at least 1,600 is found with a budget allocation of 80% software and 20% hardware. This design has a value of 1,600 and produces budget savings of 37%.

Design analysis 2: Evaluating design upgrade/-downgrade cost. In some situations, designers may have a given solution and want to know how much it will cost or save to upgrade or downgrade the solution to a different image processing accuracy. For example, designers may be asked to

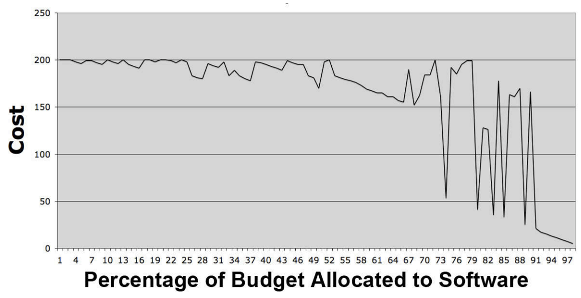


Fig. 15. Budget Surplus vs. Budget Allocation

provide a range of satellite options for their superiors that show what level of image processing accuracy they can provide at a number of price points. Figure 16 depicts another view of the ASCENT data that shows how cost varies in relation to the minimum required solution value. This graph shows that 5 cost

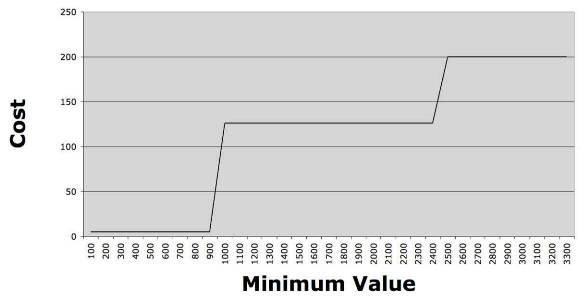


Fig. 16. Cost of Increasing Solution Value

units can finance a design with a value up to 900, but a design of a value of 1,000 units will cost at least 124 cost units. This information graph demonstrates the increased financial burden of requiring a slightly higher valued design. Alternatively, if the necessary value of the system is near the left edge of one of these plateaus, designers can make an informed decision on whether the increased value justifies the significantly increased cost.

6 RELATED WORK

Search-based software engineering has a large number of facets ranging from the design of general approximation algorithms to the construction of search-based software engineering methods for specific problems. This section compares and contrasts ASCENT to search-based software engineering techniques related to (1) approximation algorithms for solving similar problems to the MMKP co-design problem, (2) methods for using search-based techniques to solve hardware/software partitioning problems, (3) methods for using approximation techniques for solving hardware/software scheduling problems, and (4) search-based software engineering techniques for determining project staffing.

Hardware/software co-design. A number of co-design techniques [7, 24, 30, 1, 29, 34, 28, 13]—that can be viewed as search-based software engineering techniques—examine the problem of partitioning system functionality into hardware and software. These approaches use a number of search techniques ranging from binary constraint search to dynamic programming. In the partitioning problem, a system's required operations are grouped into tasks or functions, which are then implemented in either hardware or software. The goal is to correctly partition the tasks into hardware and software to meet a predefined performance goal. Some tasks may operate with higher performance if the functionality is placed on the hardware rather than on software. The performance of the system is thus determined by the location and placement of tasks in hardware versus software.

The MMKP co-design problem, which ASCENT focuses on, is complementary to this research. In particular, these related approaches do not deal with maximizing a measure of system value subject to producer/consumer resources and cost. Similarly, ASCENT does not examine the impact of the placement of tasks on the hardware and software. Each approach fills an important, although distinct, role in the search-based software engineering landscape for hardware/software co-design.

Another related problem in hardware/software co-design is the scheduling of hardware/software tasks subject to resource constraints. This type of co-design problem tries to determine the optimal ordering of a series of tasks implemented in both hardware and software. Scheduling with resource constraints is a challenging problem that has led to the development of large number of co-design search and design exploration techniques [18, 23, 14]. This co-design technique is attacking a different facet of software/-hardware co-design that does not deal with how to select a software and hardware design that maximizes system value subject to producer/consumer and cost constraints. ASCENT, however, focuses directly on this maximization of system value subject to these constraints.

MMKP approximation. Many problems similar to the hardware/software co-design problem presented in this paper have been solved with MMKP techniques. In multimedia systems, determining the quality settings that maximize the value of a series of data streams has been modeled as an MMKP problem [25, 22]. Other usages of MMKP include meta-scheduling for grid applications [32], optimally selecting design features for software product-lines [33], and book-ahead request scheduling [8]. A number of excellent heuristic approximation algorithms, such as M-HEU [3] and C-HEU [3], with near optimal results have been devised.

These existing MMKP algorithms and techniques, however, cannot be directly applied to the MMKP-codesign problem described in this paper. First, as described in Section 3.1, the existing techniques assume that there are predefined individual knapsack sizes, which is not the case in the MMKP co-design scenario. Second, as described in Section 3.2, producer MMKP items cannot be valued separately from a consumer MMKP problem, causing a coupling problem. Existing MMKP approaches are not designed to handle this type of coupling problem. In contrast, ASCENT addresses these issues and provides high-quality solutions to MMKP co-design problems.

Project management and staff allocation. Accurate planning of large projects are essential to estimate project cost, determine the formation of employee project teams, and to assign these teams to tasks in a manner that gives the largest probability for successful completion. The placement of each individual employee can change the profile of the entire project plan, resulting in an exponential number of possible configurations [5]. Moreover, parameters of a project are dynamic and may change several times before project completion, requiring that multiple staffing solutions be calculated. This research is related to MMKP co-design problems in that it deals with two tightly-coupled activites–the ordering and staffing of project parts subject to resource constraints. Although the work is related, it cannot be used to solve MMKP co-design problems. In contrast, ASCENT is specifically designed for solving MMKP co-design problems.

7 Concluding Remarks

Designing hardware and software in tandem to maximize a system capability can be an NP-hard activity. Search-based software engineering is a promising approach that can be used to leverage algorithmic techniques during system co-design. This paper presented a polynomial-time search-based software engineering technique, called *Allocation-baSed Configuration Exploration Technique* (ASCENT), for finding near optimal hardware/software co-design solutions.

We showed how ASCENT's heuristic-based solutions to hardware/software co-design problems average over 95% optimal when there are more than seven points of variability in the hardware and software design. Moreover, ASCENT's output (which is a data set showing the optimal design configurations at each ratio of budget allocation to hardware and software) can be used to search for and answer important software engineering questions, such as how much more it will cost for increasing the value of system capability.

From our experience with ASCENT, we have learned the following lessons pertaining to search-based software engineering:

- **ASCENT is amenable to parallelization.** ASCENT is highly parallelizable and amenable to multi-core architectures. Any number of budget allocation iterations can be run in parallel, allowing ASCENT to scale nearly linearly with the number of underlying computational units allocated to it.
- **CLP techniques should be used for small problems and ASCENT for large problems.** For smaller scale problems, ASCENT produces less optimal solutions. Constraint logic programming (CLP) techniques, however, work well at these small problem scales. In our experiments, roughly 7 points of variability in the hardware and software design was the cross-over point where ASCENT should be used rather than a CLP approach.
- **Some problems cannot be modeled with a single Producer/Consumer relationship.** Some problems have more than a single producer/consumer relationship. For example, when trying to simultaneously determine the configuration of an application, the underlying middleware, and the hardware there is more than one producer/-consumer relationship. For these situations, ASCENT requires breaking the problem in two and solving in phases, which is not ideal.

An implementation of ASCENT is available as part of the open-source ASCENT Design Studio project (`http://code.google.com/p/ascent-design-studio/`).

REFERENCES

[1] M. Abdelhalim and S. Habib. Modeling communication cost and hardware alternatives in PSO based HW/SW partitioning. In *Microelectronics, 2007. ICM 2007. Internatonal Conference on*, pages 111–114, 2007.

[2] M. Akbar, E. Manning, G. Shoja, and S. Khan. Heuristic Solutions for the Multiple-Choice Multi-Dimension Knapsack Problem. *International Conference on Computational Science*, pages 659–668, May 2001.

[3] M. Akbar, E. Manning, G. Shoja, and S. Khan. Heuristic Solutions for the Multiple-Choice Multi-Dimension Knapsack Problem. pages 659–668. Springer, May 2001.

[4] E. Alba and J. Francisco Chicano. Software project management with GAs. *Information Sciences*, 177(11):2380–2401, 2007.

[5] G. Antoniol, M. Di Penta, and M. Harman. A robust search-based approach to project management in the presence of abandonment, rework, error and uncertainty. *Software Metrics, 2004. Proceedings. 10th International Symposium on*, pages 172–183, 2004.

[6] A. Barreto, M. Barros, and C. Werner. Staffing a software project: A constraint satisfaction and optimization-based approach. *Computers and Operations Research*, 35(10):3073–3089, 2008.

[7] E. Barros, C. Universitaria-Recife-PE, W. Rosenstiel, and X. Xiong. A Method for Partitioning UNITY Language in Hardware and Software. *Euro-DAC'94 with Euro-VHDL'94: Proceedings, September 19-23, 1994, Grenoble, France*, 1994.

[8] P. Chiu, Y. Chen, and K. Lee. A request scheduling algorithm to support flexible resource reservations in advance. *Electrical and Computer Engineering, 2004. Canadian Conference on*, 4, 2004.

[9] L. Chung. *Non-Functional Requirements in Software Engineering*. Springer, 2000.

[10] J. Clark and J. Jacob. Protocols are programs too: the meta-heuristic search for security protocols. *Information and Software Technology*, 43(14):891–904, 2001.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT, 1990.

[12] S. Curtis. The Magnetospheric Multiscale Mission...Resolving Fundamental Processes in Space Plasmas. *NASA STI/Recon Technical Report N*, pages 48257–+, Dec. 1999.

[13] R. Dick and N. Jha. MOGAC: a multiobjective genetic algorithm for hardware-software Cosynthesis of distributed embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):920–935, 1998.

[14] C. Gebotys and M. Elmasry. *Optimal VLSI architectural synthesis: area, performance and testability*. Kluwer Academic Publishers Norwell, MA, USA, 1992.

[15] J. Gosling. *Introductory Statistics*. Pascal Press, 1995.

[16] M. Harman. The Current State and Future of Search Based Software Engineering. *International Conference on Software Engineering*, pages 342–357, 2007.

[17] M. Harman and B. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[18] P.-A. Hsiung, P.-H. Lu, and C.-W. Liu. Energy efficient co-scheduling in dynamically reconfigurable systems. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 87–92, New York, NY, USA, 2007. ACM.

[19] P. Huber, J. Wiley, and W. InterScience. *Robust statistics*. Wiley New York, 1981.

[20] T. Ibaraki, T. Hasegawa, K. Teranaka, and J. Iwase. The Multiple Choice Knapsack Problem. *J. Oper. Res. Soc. Japan*, 21:59–94, 1978.

[21] O. Ibarra and C. Kim. Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems. *Journal of the ACM (JACM)*, 22(4):463–468, 1975.

[22] M. Islam, M. Akbar, H. Hossain, and E. Manning. Admission control of multimedia sessions to a set of multimedia servers connected by an enterprise network. *Communications, Computers and signal Processing, 2005. PACRIM. 2005 IEEE Pacific Rim Conference on*, pages 157–160, 2005.

[23] M. Kim, S. Banerjee, N. Dutt, and N. Venkatasubramanian. Design space exploration of real-time multi-media mpsocs with heterogeneous scheduling policies. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 16–21, New York, NY, USA, 2006. ACM.

[24] E. Lagnese and D. Thomas. Architectural Partitioning for System Level Design. *Design Automation, 1989. 26th Conference on*, pages 62–67, 1989.

[25] A. Lawabni and A. Tewfik. Resource Management and Quality Adaptation in Distributed Multimedia Networks. *Proceedings of the 10th IEEE Symposium on Computers and Communications (ISCC'05)-Volume 00*, pages 604–610, 2005.

[26] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification & Reliability*, 14(2):105–156, 2004.

[27] L. Northrop, P. Feiler, B. Pollak, and D. Pipitone. *Ultra-large-scale Systems: The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon University, 2006.

[28] D. Saha, R. Mitra, and A. Basu. Hardware software partitioning using genetic algorithm. In *VLSI Design, 1997. Proceedings., Tenth International Conference on*, pages 155–160, 1997.

[29] Q. Tong, X. Zou, Q. Zhang, F. Gao, and H. Tong. The Hardware/Software Partitioning in Embedded System by Improved Particle Swarm Optimization Algorithm. In *Fifth IEEE International Symposium on Embedded Computing, 2008. SEC'08*, pages 43–46, 2008.

[30] F. Vahid, D. Gajski, and J. Gong. A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning. *Proceedings of the conference on European design automation*, pages 214–219, 1994.

[31] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Constraint-Based Reasoning*, 1994.

[32] D. Vanderster, N. Dimopoulos, and R. Sobie. Metascheduling Multiple Resource Types using the MMKP. *Grid Computing, 7th IEEE/ACM International Conference on*, pages 231–237, 2006.

[33] J. White, B. Dougherty, and D. Schmidt. Filtered Cartesian Flattening. *Workshop on Analysis of Software Product-Lines at the International Conference on Software Product-lines*, October 2008.

[34] T. Wiangtong, P. Cheung, and W. Luk. Comparing three heuristic search methods for functional partitioning in hardware–software codesign. *Design Automation for Embedded Systems*, 6(4):425–449, 2002.