Automating Hardware and Software Evolution Analysis

Brian Dougherty, Jules White, Chris Thompson, and Douglas C. Schmidt
Department of Electrical Engineering and Computer Science,
Vanderbilt University, Nashville, TN, USA
{briand,jules,schmidt}@dre.vanderbilt.edu & chris.m.thompson@vanderbilt.edu

Abstract

Cost-effective software evolution is critical to many distributed real-time and embedded (DRE) systems. Selecting the lowest cost set of software components that meet DRE system resource constraints, such as total memory and available CPU cycles, is an NP-Hard problem. This paper provides three contributions to R&D on evolving software-intensive DRE systems. First, we present the Software Evolution Analysis with Resources (SEAR) technique that transforms component-based DRE system evolution alternatives into multidimensional multiple-choice knapsack problems. Second, we compare several techniques for solving these knapsack problems to determine valid, low-cost design configurations for resource constrained component-based DRE systems.. Third, we empirically evaluate the techniques to determine their applicability in the context of common evolution scenarios. Based on these findings, we present a taxonomy of the solving techniques and the evolution scenarios that best suit each technique.

1 Introduction

Current trends and challenges. Evolution accounts for a significant portion of software life-cycle costs [14]. An important type of software evolution involves enhancing existing software to meet new customer and market needs [9]. For example, in the automotive industry, each year the software and hardware from the previous year's model car must be upgraded to provide new capabilities, such as automated parking or wireless connectivity.

Software evolution analysis is the process of determining which software components and hardware components can be added to a system to implement new functionality while adhering to multiple resource constraints. This analysis involves several challenges, including (1) building an economic model to estimate the return on investment of new software features, (2) estimating the cost of implementing a software feature [13], and (3) selecting a new system configuration that maximizes the value of the features added while respecting resource constraints. This paper examines software evolution analysis techniques that automatically determine valid system configurations that support required new capabilities without violating resource constraints.

In many domains, the cost/benefit analysis for software evolution is partially simplified by the availability of commercial-off-the-shelf (COTS) software/hardware components [10]. For example, automotive manufacturers know how much it costs to buy windshield wiper hardware/software components, as well as electronic control units (ECUs) with specific memory and processing capabilities/costs. Similar cost/benefit analysis can also be conducted for custom-developed (*i.e.*, non-COTS) software/hardware components [3].

Regardless of whether components are COTS or custom, however, determining the optimal subset of components needed to upgrade existing components is an NP-Hard problem [4]. In the simplest case—where any combination of the components are compatible—the problem of selecting which components to use in an upgrade is an instance of the *knapsack problem*, where a knapsack of predefined size is filled with items of various sizes and values. The goal is to maximize the sum of the value of items in the sack without exceeding the knapsack size. In this paper, the knapsack size is defined by the total budget available for the component purchase and/or development; the goal is to find the optimal subset of the hardware and software components that do not exceed the budget (*i.e.*, that fit into the knapsack) and maximize the value of the added capabilities [12].

Moreover, many software evolution problems do not fit into a relaxed paradigm where any set of components can be used. For example, purchasing two different infotainment software system implementations does not double the value of the car since only one system can actually be installed. In most situations, each new capability that can be added is a point of *design variability*, with a number of potential implementations [18] each having their own cost and performance. The infotainment system is the point of design variability and the various implementations of the infotainment system are the concrete options for that point of variability since only one concrete option can be chosen at one point in time.

Distributed, real-time, and embedded (DRE) systems, such as automotive and avionics systems, have limited resources and often exhibit tight coupling between hardware and software decisions [17]. A consequence of this tight-coupling is that the selected hardware components must provide sufficient resources to support the decisions made for the points of software variability. For example, purchasing an infotainment software system that consumes more memory than is available on its hosting hardware can yield a flawed configuration. When determining the set of software components to upgrade, therefore, careful consideration must be paid to the production and consumption of resources by hardware and software, respectively. Finding the set of replacement components that adheres to all resource constraints and maximizes total value for an upgrade is an *optimization problem* that focuses on determining solution(s) that maximizes a single element of the problem.

This type of hardware/software co-design problem is NP-Hard [19] since there are an exponential number of possible evolved configurations, which prohibits the use of exhaustive state space exploration even for minor DRE system software evolution. For example, Consider the evolution of an automobile braking system with 10 different points of software variability and 10 implementation options for each variability point. Likewise, assume there is a single variable hardware electronic control units (ECU) with 10 different available configurations with varying capabilities. This problem formulation has 10^{100} possible evolution configurations that must be considered.

Solution Approach→MMKP-based upgrade analysis. This paper shows how a number of complex software evolution optimization problems can be recast as *multidimensional multiple-choice knapsack problems* (MMKP) [11]. MMKPs are a specialized version of the more general knapsack problem where the items are divided into sets and exactly one item must be chosen from each set. The goal of the MMKP problem is to maximize the value of the items placed into the knapsack without violating the knapsack size or set selection constraints.

By converting the task of determining valid, favorable software evolution configurations into an instance of an MMKP, developers can take advantage of powerful approximation algorithms [6]. While these algorithms do not guarantee optimal solutions, they frequently find near-optimal solutions in polynomial-time. Moreover, certain software evolution analysis problems that involve tightly-coupled hardware/software decisions can be framed as co-dependent MMKP problems, in which one problem produces resources for consumption by the other [19].

Converting software evolution decisions into tractable instances of MMKP problems is neither obvious nor trivial. This process is exacerbated by DRE systems in which software architecture decisions may effect the hardware architecture and vice versa, thereby complicating the evolution analysis. This paper provides the following four contributions to the study of techniques that convert various software evolution analysis problems into MMKP instances: (1) we describe the *Software Evolution Analysis with Resources* (SEAR) technique for mapping software evolution analysis problems of several common software evolution scenarios to MMKP problems, (2) we show how these MMKP formulations of software evolution analysis problems can be solved using MMKP heuristic techniques and mapped back to upgrade solutions, (3) we present empirical comparisons of the optimality and solve times for three algorithms that can solve these transformed MMKP problems for problems of various size, and (4) we used this data to present a taxonomy that describes which algorithm is most effective based on the problem type and size.

Paper organization. The remainder of this paper is organized as follows: Section 2 presents the automotive evolution example used to showcase the need for—and applicability of—our SEAR technique; Section 3 describes several challenge problems to which SEAR can be applied; Section 4 qualitatively evaluates applying SEAR to these challenge problems and Section 5 then quantitatively evaluates applying SEAR to these challenge problems; Section 6 compares SEAR with related work; Finally, Section 7 summarizes our findings and presents lessons learned.

2 Motivating Example

It is hard to upgrade the software and hardware in a DRE system to support new software features *and* adhere to resource constraints. For example, auto manufacturers that want to integrate automated parking software into a car must find a way to upgrade the hardware on the car to provide sufficient resources for the new software. Each automated parking software package may need a distinct set of controllers for movement (such as brake and throttle) and ECU processing capabilities (such as memory and processing power) [5].

Figure 1 shows a segment of automotive software and hardware design that we use as a motivating example throughout the paper. This legacy configuration contains two software components: a legacy brake controller and a legacy throttle controller as shown in Figure 1(A). In addition to an associated value and purchase cost, each component consumes memory and processing power to function. These resources are provided by the hardware component (*i.e.*, the ECU). This configuration is valid since the ECU produces more memory and processing resources than the components collectively require.

Adding an automated parking system to the original design shown in Figure 1(A) may require software components that are more recent, more powerful, or provide more functionality than the original software components. For example, to provide automated parking, the throttle controller may need to possess functionality to interface with laser depth sensors. In this example, the original controller lacked this functionality and must be upgraded with a more

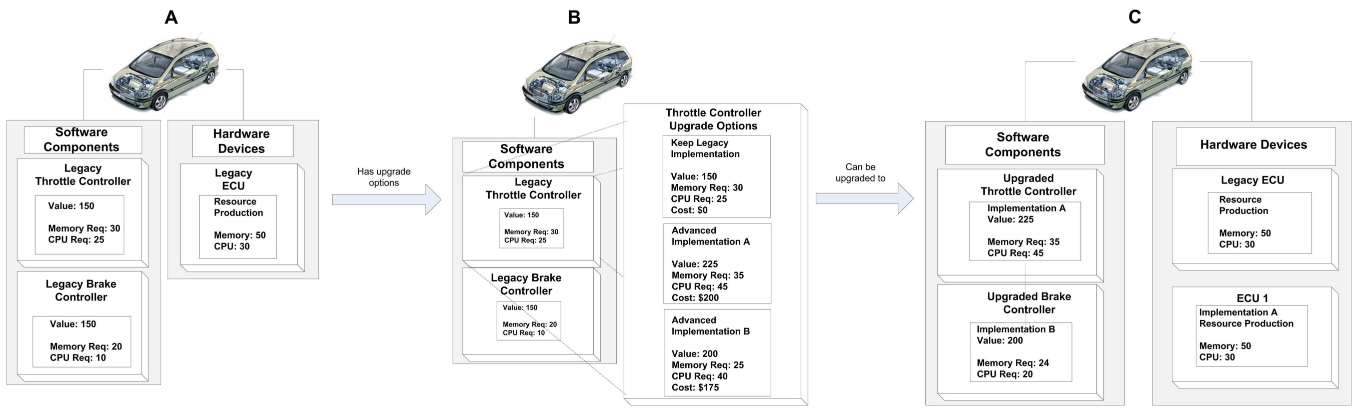

Figure 1. Software Evolution Progression

advanced implementation. The implementation options for the throttle controller are shown in Figure 1(B).

Figure 1(B) shows potential controller evolution options. Two implementations are available for each controller.Developers installing an automated parking system must upgrade the throttle controller via one of the two available implementations and can optionally increase the functionality of the system by upgrading the brake controller.

Given a fixed software budget (*e.g.*, $500), developers can purchase any combination of controllers. If developers want to purchase both a new throttle controller *and* a new brake controller, however, they must purchase an additional ECU to provide the necessary resources. The other option is to not upgrade the brake controller, thereby sacrificing additional functionality, but saving money in the process.

Given a fixed total hardware/software budget of $700, the developers must first divide the budget into a hardware budget and a software budget. For example, they could divide the budget evenly, allocating $350 to the hardware budget and $350 to the software budget. With this budget developers can afford to upgrade the throttle controller software with Implementation B and the brake controller software with Implementation B. The legacy ECU alone, however, does not provide enough resources to support these two devices. Developers must therefore purchase an additional ECU to provide the necessary additional resources. The new configuration for this segment of the automobile with upgraded controllers and an additional ECU (with ECU1 Implementation A) can be seen in Figure 1(C).

Our motivating example above focused on 2 points of software design variability that could be implemented using 4 different new components. Moreover, 4 different potential hardware components could be purchased to support the software components. To derive a configuration for the entire automobile, an additional 46 software components and 20 other hardware components must be examined. Each configuration of these components could be a valid configuration, resulting in (50^{24}) unique potential configurations. In general, as the quantity of software and hardware options increase, the number of possible configurations increases exponentially, thereby rendering manual optimization solutions infeasible in practice.

3 Challenges of Evolution Decision Analysis

Several issues must be addressed when evolving software and hardware components. For example, developers must determine (1) what software and hardware components to buy and/or build to implement the new feature, (2) how much of the total budget to allocate to software and hardware, respectively, and (3) that the selected hardware components provide sufficient resources for the chosen software components. These issues are related, *e.g.*, developers can either choose the software and hardware components to dictate the allocation of budget to software and hardware or the budget distributions can be fixed and then the components chosen. Moreover, developers can either choose the hardware components and then select software features that fit the resources provided by the hardware or the software can be chosen to determine what resource requirements the hardware must provide. This section describes a number of challenging upgrade scenarios that require developers to address the issues outlined above.

3.1 Challenge 1: Evolving Hardware to Meet New Software Resource Demands

This evolution scenario has no variability in implementing new functionality, *i.e.*, the set of software resource requirements is predefined. For example, if an automotive manufacturer has developed an in-house implementation of an automated parking system, the manufacturer will know the new hardware resources needed to support the system and must determine which hardware components to purchase from vendors to satisfy the new hardware requirements. Since the only purchases that must be made are for hardware, the exact budget available for hardware is known. The problem is to find the least-cost hardware design that can provide the resources needed by the software.

The difficulty of this scenario can be shown by assuming that there are 10 different hardware components that can be evolved, resulting in 10 points of hardware variability. Each replaceable hardware component has 5 implementation options from which the single upgrade can be chosen, thereby creating 5 options for each variability point.

To determine which set of hardware components yield the optimum value (*i.e.*, the highest expected return on investment) or the minimum cost (*i.e.*, minimum financial budget required to construct the system), 9,765,265 configurations of component implementations must be examined. Even after each configuration is constructed, developers must determine if the hardware components provides sufficient resources to support the chosen software configuration. Section 4.1 describes how SEAR addresses this challenge by using predefined software components and replaceable hardware components to form a single MMKP evolution problem.

3.2 Challenge 2: Evolving Software to Increase Overall System Value

This evolution scenario preselects the set of hardware components and has no variability in the hardware implementation. Since there is no variability in the hardware, the amount of each resource available for consumption is fixed. The software components, however, must be evolved. For example, a software component on a common model of automobile has been found to be defective. To avoid the cost of a recall, the manufacturer can ship new software components to local dealerships, who can replace the defective software components. The dealerships lack the capabilities required to add hardware components to the automobile.

Since no new hardware is being purchased, the entire budget can be devoted to software purchases. As long as the resource consumption of the chosen software component configuration does not exceed the resource production of existing hardware components, the configuration can be considered valid. The difficulty of this challenge is similar to the one described in Section 3.1, where 10 different types of software components with 5 different available selections per type required the analysis of 9,765,265 configurations. Section 4.2 describes how SEAR addresses this challenge by using the predetermined hardware components a and evolution software components to create a single MMKP evolution problem.

3.3 Challenge 3: Unrestricted Upgrades of Software and Hardware in Tandem

Yet another challenge occurs when both hardware components and software components can be added, removed, or replaced. For example, consider an automobile manufacturer designing the newest model of its flagship sedan. This sedan could either be similar to the previous model with few new software and hardware components or it could be completely redesigned, with most or all of the software and hardware components evolved.

Though the total budget is predefined for this scenario, it is not partitioned into individual hardware and software budgets, thereby greatly increasing the magnitude of the problem. Since neither the total provided resources nor total consumable resources are predefined, the software components depend on the hardware decisions and vice versa, incurring a strong coupling between the two seemingly independent MMKP problems.

The solution space of this problem is even larger than that of the challenge presented in Section 3.2. Assuming there are 10 different types of hardware options with 5 options per type, there are 9,765,265 possible hardware configurations. In this case, however, every type of software is eligible instead of just the types that are to be upgraded. If there are 15 types of software with 5 options per type, therefore, 30,516,453,125 software variations can be chosen. Each variation must be associated with a hardware configuration to test validity, resulting in 30,516,453,125 * 9,765,265 tests for each budget allocation.

In these worst case scenarios, the staggering size of the configuration space prohibits the use of exhaustive search algorithms for anything other than trivial design problems. Section 4 describes how SEAR addresses this challenge by combining all software and hardware components into a specialized MMKP evolution problem.

4 Evolution Analysis via SEAR

This section describes the procedure for transforming the evolution scenarios presented in Section 3 into evolution *Multidimensional Multiple-choice Knapsack Problems* (MMKP) [2]. MMKP problems are appropriate for representing evolution scenarios that comprise a series of points of design variability that are constrained by multiple resource constraints, such as the scenarios described in Section 3. In addition, there are several advantages to mapping the scenarios to MMKP problems.

MMKP problems have been extensively studied. As a result, there are several polynomial time algorithms that can be utilized to provide nearly optimal solutions, such as those described in [15, 7, 2, 8]. This paper uses the M-HEU approximation algorithm described in [2] for evolution problems with variability in either hardware or software but not both. The multidimensional nature of MMKP problems is ideal for enforcing multiple resource constraints. The multiple-choice aspect of MMKP problems make them appropriate for situations (such as those described in Section 3.2) where only a single software component implementation can be chosen for each point of design variability.

MMKP problems can be used to represent situations where multiple options can be chosen for implementation. Each implementation option consumes various amounts of resources and has a distinct value. Each option is placed into a distinct MMKP set with other competing options and only a single option can be chosen from each set. A valid configuration results when the combined resource consumption of the items chosen from the various MMKP sets does not exceed the resource limits. The value of the solution is computed as the sum of the values of selected items.

4.1 Mapping Hardware Evolution Problems to MMKP

Below we show how we map the hardware evolution problem described in Section 3.1 to an MMKP problem. In this case, the scenario can be mapped to a single MMKP problem representing the points of hardware variability. The size of the knapsack is defined by the hardware budget. The only additional constraint on the MMKP solution is that the quantities of resources provided by the hardware configuration exceeds the predefined consumption needs of software components.

To create the hardware evolution MMKP problem, each hardware component is converted to an MMKP item. For each point of hardware variability, an MMKP set is created. Each set is then populated with the MMKP items corresponding to the hardware components that are implementation options for the set's corresponding point of hardware variability. Figure 2 shows a mapping of a hardware evolution problem for an ECU to an MMKP.

In Figure 2 the software does not have any points of variability that are eligible for evolution. Since there is no variability in the software, the exact amount of each resource that will be consumed by the software is known. The M-HEU approximation algorithm (or an exhaustive search algorithm, such as a linear constraint solver) uses this hardware evolution MMKP problem, the predefined resource consumption, and the predefined external resource (budget) requirements to determine which ECUs to purchase and install. The solution to the MMKP is the hardware components that should be chosen to implement each point of hardware variability.

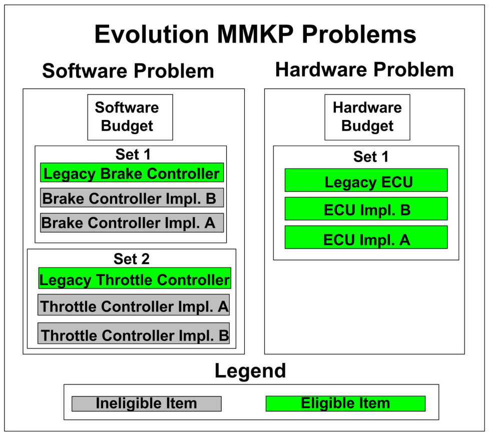

Figure 2. MMKP Representation of Hardware Evolution Problem

4.2 Mapping Software Evolution Problems to MMKP

We now show how to map the software evolution problem described in Section 3.2 to an MMKP problem. In this case, the hardware configuration cannot be altered, as shown in Figure 3. The hardware thus produces a predeter-

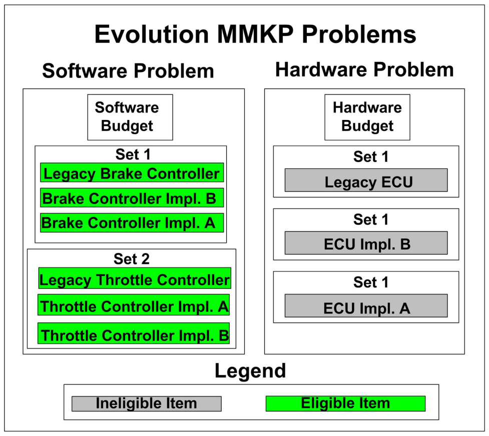

Figure 3. MMKP Representation of Software Evolution Problem

mined amount of each resource. Similar to Section 4.1, the fiscal budget available for software purchases is also predetermined. Only the software evolution MMKP problem must therefore be solved to determine an optimal solution.

As shown in the *software problem* portion of Figure 3, each point of software variability becomes a set that contains the corresponding controller implementations. For each set there are multiple implementations that can serve as the controller. This software evolution problem—along with the software budget and the resources available for consumption as defined by the hardware configuration—can be used by an MMKP algorithm to determine a valid selection of throttle and brake controllers.

4.3 Hardware/Software Co-Design with ASCENT

Several approximation algorithms can be applied to solve single MMKP problems, as described in Sections 4.1 and 4.2. These algorithms, however, cannot solve cases in which there are points of variability in both hardware and software that have eligible evolution options. In this situation, the variability in the production of resources from hardware and the consumption of resources by software requires solving two MMKP problems simultaneously, rather than one. In prior work we developed the *Allocation-baSed Configuration Exploration Technique* (ASCENT) technique to determine valid, low-cost solutions for these types of dual MMKP problems [19].

ASCENT is a search-based, hardware/software co-design approximation algorithm that maximizes the software value of systems while ensuring that the resources produced by the hardware MMKP solution are sufficient to support the software MMKP solution [19]. The algorithm can be applied to system design problems in which there are multiple producer/consumer resource constraints. In addition, ASCENT can enforce external resource constraints, such as adherence to a predefined budget.

The software and hardware evolution problem described in Section 3.3 must be mapped to two MMKP problems so ASCENT can solve them. The hardware and software evolution MMKP problems are prepared as shown in Figure 4. This evolution differs from the problems described in Section 4.1, since all software implementations are now eligible

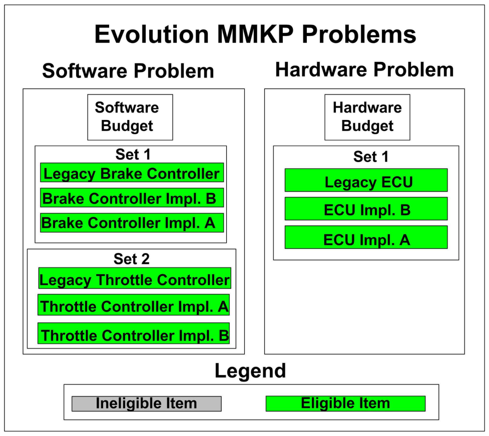

Figure 4. MMKP Representation of Unlimited Evolution Problem

for evolution, thereby dramatically increasing the amount of variability. These two problems—along with the total budget—are passed to ASCENT, which then searches the configuration space at various budget allocations to determine a configuration that optimizes a linear function computed over the software MMKP solution. Since ASCENT utilizes an approximation algorithm, the total time to determine a valid solution is usually small. In addition, the solutions it produces average over 90% of optimal [19].

5 Empirical Results

This section presents empirical data obtained from using three different algorithmic techniques to determine valid, high-value, evolution configurations for the scenarios described in Section 3. These results demonstrate that each algorithm is effective for certain types of MMKP problems. Moreover, if the correct technique is used, a near-optimal solution can be found. Each set represents a point of design variability and problems with more sets have more variability. Moreover, the ASCENT and M-HEU algorithms can be used to determine solutions for large-scale problems that cannot be solved in a feasible amount of time with exhaustive search algorithms.

5.1 Experimental Platform

All algorithms were implemented in Java and all experiments were conducted on an Apple MacbookPro with a 2.4 GHz Intel Core 2 Duo processor, 2 gigabytes of RAM, running OS X version 10.5.5, and a 1.6 Java Virtual Machine (JVM) run in client mode. For our exhaustive MMKP solving technique—which we call the linear constraint solver (LCS)—we used a branch and bound solver built on top of the Java Choco Constraint Solver (`choco.sourceforge.net`). The M-HEU heuristic solver was a custom implementation that we developed with Java. The ASCENT algorithm was also based on a custom implementation with Java.

5.2 Hardware Evolution with Predefined Resource Consumption

This experiment investigates the use of a linear constraint solver and the use of the M-HEU algorithm to solve the challenge described in Section 3.1, where the software components are fixed. First, we test for the total time needed for each algorithm to run to completion. We then examine the optimality of the solutions generated by each algorithm. We run these tests for several problems with increasing set counts, thus showing how each algorithm performs with increased design variability.

Figure 5 shows the time required to generate a hardware configuration if the software configuration is predefined.[1]

[1]Time is plotted on a logarithmic scale for all figures that show solve time.

Since only a single MMKP problem must be solved, we

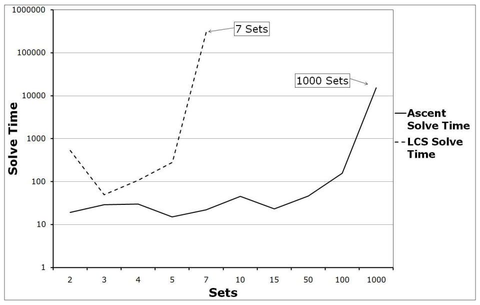

Figure 5. Solve Time vs Number of Sets

use the M-HEU algorithm. As set size increases, the time required for the linear constraint solver increases rapidly. If the problem consists of more sets, the time required for the linear constraint solver becomes prohibitive. The M-HEU approximation algorithm, however, scaled much better, finding a solution for a problem with 1,000 sets in ∼15 seconds.

Figure 6 shows that both algorithms generated solutions with 100% optimality for problems with 5 or less sets. Regardless of the number of sets, the M-HEU algorithm

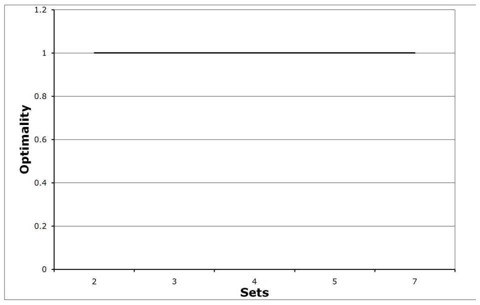

Figure 6. Solution Optimality vs Number of Sets

completed faster than the linear constraint solver without sacrificing optimality.

5.3 Software Evolution with Predefined Resource Production

This experiment examines the use of a linear constraint solver and the M-HEU algorithm to solve evolution scenarios in which the hardware components are fixed, as described in Section 3.2. We test for the total time each algorithm needs to run to completion and examine the optimality of solutions generated by each algorithm.

Figure 7 shows the time required to generate a software configuration generated if the hardware configuration is predetermined. As with Challenge 2, the M-HEU algorithm is

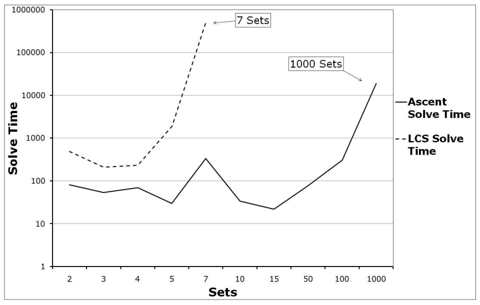

Figure 7. Solve Time vs Number of Sets

used since only a single MMKP problem must be solved. Once again, LCS's limited scalability is demonstrated since the required solve time makes its use prohibitive for problems with more than five sets. The M-HEU solver scales considerably better and can solve a problem with 1,000 sets in less than 16 seconds, which is fastest for all problems.

Figure 8 shows the optimality provided by each solver. In this case, the M-HEU solver is only 80% optimal for

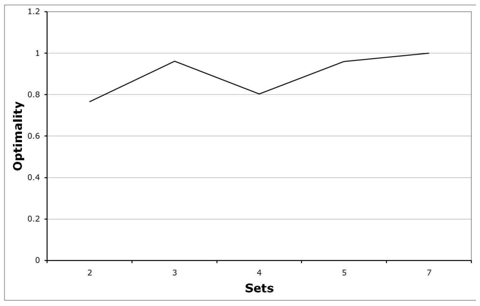

Figure 8. Solution Optimality vs Number of Sets

problems with 4 sets. Fortunately, the optimality improves with each increase in set count with a solution for a problem with 7 sets being 100% optimal.

5.4 Unrestricted Software Evolution with Additional Hardware

This experiment examines the use of a linear constraint solver and the ASCENT algorithm to solve the challenge described in Section 3.3, in which no hardware or software components are fixed. We first test for the total time needed for each algorithm to run to completion and then examine the optimality of the solutions generated by each algorithm. Unrestricted evolution of software and hardware components has similar solve times to the previous experiments.

Figure 9 shows that regardless of the set count for the MMKP problems, the ASCENT solver derived a solution much faster than LCS. This figure also shows that the re-

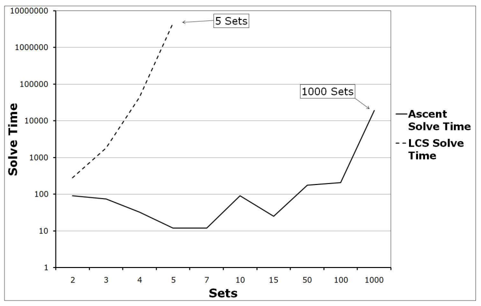

Figure 9. Solve Time vs Number of Sets

quired solve time to determine a solution with LCS increases rapidly, *e.g.*, problems that have more than five sets require an extremely long solve time. The ASCENT algorithm once again scales considerably better and can even solve problems with 1,000 or more sets. In this case, the optimality of the solutions found by ASCENT is low for problems with 5 sets, as shown in Figure 10. Fortunately, the

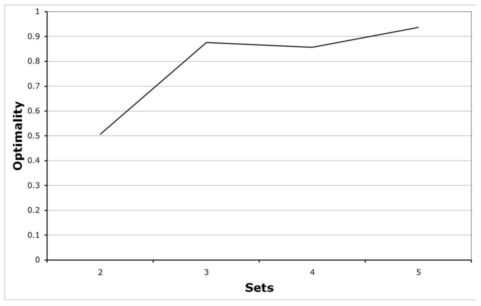

Figure 10. Solution Optimality vs Number of Sets

time required to solve with LCS is not prohibitive in these cases, so it is still possible to find a solution with 100% optimality in a reasonable amount of time.

5.5 Comparison of Algorithmic Techniques

this experiment compared the performance of LCS to the performance of the M-HEU and ASCENT algorithms for all challenges in Section 3. As shown in Figure 11, the characteristics of the problem(s) being solved has a large impact on solving duration.

Each challenge has more points of variability than the previous challenge. The solving time for LCS thus increases as the number of the points of variability increases. For all cases, the LCS algorithm requires an exorbitant amount of time for problems with more than five sets. In contrast, the M-HEU and ASCENT algorithms show no discernable correlation between the amount of variability and

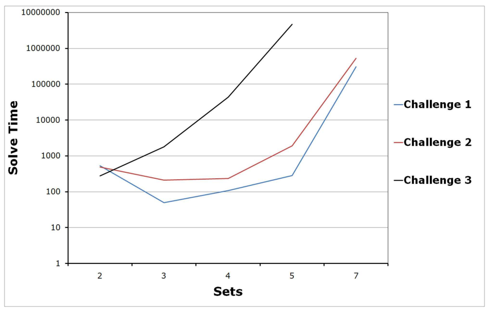

Figure 11. LCS Solve Times vs Number of Sets

the solve time. In some cases, problems with more sets require more time to solve than problems with less sets, as shown in Figure 12.

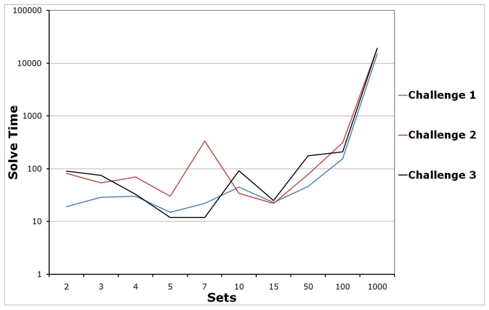

Figure 12. M-HEU & ASCENT Solve Times vs Number of Sets

Figure 13 compares the scalability of the three algorithms. This figure shows that LCS requires the most solv-

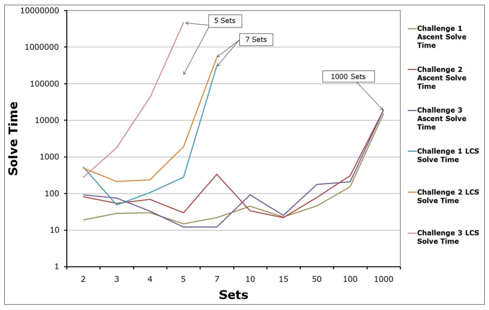

Figure 13. Comparison of Solve Times for All Experiments

ing time in all cases. Likewise, the ASCENT and M-HEU algorithms scale at approximately the same rate for all problems and are far superior to the LCS algorithm. The optimality of the ASCENT and M-HEU algorithms is near-optimal only for problems with five or more sets, as shown in Figure 14. The exception to this trend occurs if there

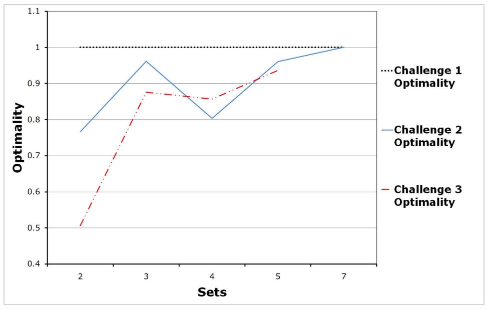

Figure 14. Comparison of Optimalities for All Experiments

are few points of variability, *e.g.*, when there are few sets and the software is predetermined. These findings motivate the taxonomy shown in Figure 15 that describes which algorithm is most appropriate, based on problem size and variability.

Solver	Variability in Either Hardware or Software		Variability in Both Hardware and Software	
	Sets ≥ 8	Sets < 8	Sets ≥ 6	Sets < 6
LCS		X		X
M-HEU	X			
ASCENT			X	

Figure 15. Taxonomy of Techniques

6 Related Work

This section compares/contrasts the strategy used by SEAR for evolution analysis with the use of (1) architecture reconfigurations to satisfy multiple resource constraints and (2) resource planning in enterprise organizations to facilitate upgrades.

Architectual considerations of embedded systems. Many hardware/software co-design techniques can be used to analyze the effectiveness of embedded system architectures. Slomka et al [16] discuss the development life cycle of designing embedded systems. In their approach, various partitionings of software onto hardware devices are proposed and analyzed to determine if predefined performance requirements can be met. If the performance goals are not attained, the architecture of the system will be modified by altering the placement of certain devices in the architecture. Even if a valid configuration is determined, it may still be possible to optimize the performance by moving devices.

While optimization is an integral application of SEAR, it is not achieved by altering the system architecture. The only choices that can affect the system performance and value is the choice of which type of hardware and/or software component to perform the functionality defined in the architectural design. Moreover, architectural hardware/software co-design decisions traditionally do not consider comparative resource constraints or financial cost optimization.

Maintenance models for enterprise organizations. The difficulty of software evolution is a common and significant obstacle in business organizations. Ng et al [13] discuss the impact of vendor choice and hardware consumption to show the sizable financial and functional impact that results from installing *enterprise resource planning* (ERP) software. Other factors related to calculating evolution costs include vendor technical support, the difficulty of replacing the previous version of the software, and annual maintenance costs. Maintenance models are used to predict and plan the effect of purchasing and utilizing various software options on overall system value. Steps for the creating maintenance models with increased accuracy for describing the ramifications of an ERP decision are also presented.

Currently, maintenance models require a substantial amount of effort to calculate the overall impact of installing a single software package, much of which can not be done through computation. SEAR analyzes the plausibility and impact of deploying many software components onto multiple hardware devices. While maintenance models can be used to assess the value of the functionality and durability added by a certain software package, they have not been used to explore the hardware/software co-design space to determine valid configurations from large sets of potential hardware devices and software components. Instead, they are used to define a process for analyzing and calculating the value of predefined upgrades. SEAR is used to solve the complex problem of determining determine valid evolution configurations. Only after the discovery of these configurations can ERPs be used to predict the overall impact of their installation.

7 Concluding Remarks

Determining valid evolution configurations for software/hardware configurations that increase system value is hard. The exponential number of possible configurations that stem from the massive variability in these problems prohibit the use of exhaustive search algorithms for non-trivial problems. This paper presented the *Software Evolution Analysis with Resources* (SEAR) technique, which converts common evolution problems into *multi-dimensional multiple-choice knapsack problems* (MMKP). We also empirically evaluated three different algorithms for solving these problems to compare their effectiveness in providing valid, high-value evolution configurations.

From these experiments, we learned the following lessons pertaining to determine valid evolution configurations for hardware/software co-design systems:

• **Approximation algorithms scale better than exhaustive algorithms.** Exhaustive search techniques, such as the linear constraint solver algorithm, cannot be applied to non-trivial problems. The determining factor in the effectiveness of these algorithms is the number of problem sets. To solve problems with realistic set counts in feasible time, approximation algorithms, such as the M-HEU algorithm or the ASCENT algorithm must be used. These techniques can solve even large problems in seconds, with minimal impact on optimality.

• **Extremely small or large problems yield near-optimal solutions.** For non-trivial problems, the ASCENT algorithm and M-HEU algorithm can be used to determine near-optimal evolution configurations. For tiny problems, the LCS algorithm can be used to determine optimal solutions. Given that these tiny problems have few points of variability, this can be done rapidly.

• **Problem size should determine which algorithm to apply.** Based on problem characteristics, it can be highly advantageous to use one algorithmic technique versus another, which can result in faster solving times or higher optimality. Figure 15 shows the problem attributes that should be examined when deciding which algorithm to apply. It also relates the algorithm that is best suited for solving these evolution problems based on the number of sets present.

• **No algorithm is universally superior.** Based on the analysis of empirical results, we determined that all three algorithms are superior for different types of evolution problems. We have not, however, discovered an algorithm that performs well for every problem type. To determine if other existing algorithms perform better for one or all types of evolution problems, further experimentation and analysis is necessary. Our future work will examine other approximation algorithms, such as genetic algorithms and particle swarm techniques, to determine if a single superior algorithm exists.

The current version of ASCENT with example code that utilizes SEAR is available in open-source form at `ascent-design-studio.googlecode.com`.

References

[1] M. Akbar, E. Manning, G. Shoja, and S. Khan. Heuristic Solutions for the Multiple-Choice Multi-dimension Knapsack Problem. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 659–668, 2001.

[2] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby. Cost models for future software life cycle processes: COCOMO 2.0. *Annals of Software Engineering*, 1(1):57–94, 1995.

[3] X. Gu, P. Yu, and K. Nahrstedt. Optimal Component Composition for Scalable Stream Processing. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*, pages 773–782, 2005.

[4] J. Her, S. Choi, D. Cheun, J. Bae, and S. Kim. A Component-Based Process for Developing Automotive ECU Software. *LECTURE NOTES IN COMPUTER SCIENCE*, 4589:358, 2007.

[5] M. Hifi, M. Michrafy, and A. Sbihi. Heuristic algorithms for the multiple-choice multidimensional knapsack problem. *Journal of the Operational Research Society*, 55(12):1323–1332, 2004.

[6] M. Hifi, M. Michrafy, and A. Sbihi. A Reactive Local Search-Based Algorithm for the Multiple-Choice Multi-Dimensional Knapsack Problem. *Computational Optimization and Applications*, 33(2):271–285, 2006.

[7] C. Hiremath and R. Hill. New greedy heuristics for the Multiple-choice Multi-dimensional Knapsack Problem. *International Journal of Operational Research*, 2(4):495–512, 2007.

[8] A. Jost and A. Franke. Residual Value Analysis. 2005.

[9] G. Leen and D. Heffernan. Expanding Automotive Electronic Systems. *Computer*, 35(1):88–93, 2002.

[10] E. Lin. A Biblographical Survey on Some Wellknown Non-Standard Knapsack Problems. *INFOR-OTTAWA-*, 36:280–283, 1998.

[11] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. 1990.

[12] C. Ng and G. Chan. An ERP maintenance model. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, page 10, 2003.

[13] S. Schach. *Classical and Object-Oriented Software Engineering*. McGraw-Hill Professional, 1995.

[14] A. Shahriar, M. Akbar, M. Rahman, and M. Newton. A multiprocessor based heuristic for multi-dimensional multiple-choice knapsack problem. *The Journal of Supercomputing*, 43(3):257–280, 2008.

[15] F. Slomka, M. Dorfel, R. Munzenberger, and R. Hofmann. Hardware/software codesign and rapid prototyping of embeddedsystems. *Design & Test of Computers, IEEE*, 17(2):28–38, 2000.

[16] S. Srinivasan and N. Jha. Hardware-software co-synthesis of fault-tolerant real-time distributed embedded systems. In *European Design Automation Conference: Proceedings of the conference on European design automation*, volume 18, pages 334–339, 1995.

[17] N. Ulfat-Bunyadi, E. Kamsties, and K. Pohl. Considering Variability in a System Familys Architecture During COTS Evaluation. In *Proceedings ofthe 4th International Conference on COTS-Based Software Systems (ICCBSS 2005), Bilbao, Spain*. Springer.

[18] J. White, B. Dougherty, and D. C. Schmidt. Ascent: An algorithmic technique for designing hardware and software in tandem. Technical Report ISIS-08-907, ISIS-Vanderbilt University, August 2008.