Fast Design Space Exploration Through Validity and Quality Filtering of Subsystem Designs

Santosh G. Abraham, B. Ramakrishna Rau, Robert Schreiber
Compiler and Architecture Research
HP Laboratories Palo Alto
HPL-2000-98
July 25th, 2000*

E-mail: {abraham,rau, schreiber}@hpl.hp.com

automated design, multi-objective optimization, multiple criteria optimization, design space decomposition, hierarchical evaluation, performance evaluation, trace-driven simulation

Automated design tools help to capture the benefits of customization in embedded system design while not exceeding design budgets. Such design tools must understand and exploit the hierarchical structure of design spaces, because systems of any significant complexity typically consist of components (subsystems). In order to reduce the design cost for such systems, designers develop the best component designs and restrict the number of system designs that they evaluate to those formed by combining these component designs.

This report formalizes this overall approach. A decomposition splits a system design space into smaller design spaces, one for each of the components. The validity function determines whether a particular composition of component designs is a valid system design. We develop efficient ways to filter component designs from further consideration with minimal or no effect on the quality of system designs. First, we use a component-level validity filter to exclude component designs that are not part of any system design. Second, we split each of the remaining component designs into smaller predicated spaces, such that designs from only certain combinations of these spaces generate all valid designs. Finally, we employ component-level evaluation functions to generate high quality component designs. We show how these results were applied in the automated design of an embedded system consisting of a processor, hardware accelerator and cache memory hierarchy.

* Internal Accession Date Only Approved for External Publication

1 Introduction

Embedded computer systems are used in a wide range of devices and appliances such as mobile phones, printers, and cars. Though these appliances are not usually regarded as computer systems, they rely heavily on embedded computer systems to provide key functionality and usability features. In many cases, the required computing capabilities of embedded systems match or even exceed those in general-purpose computers, due to stringent real-time data handling needs. Furthermore, embedded systems must often meet severe cost and power dissipation requirements. The number of embedded computers in these appliances far exceeds the number of general-purpose computer systems such as PCs or servers. In the future, the revenue stream from these embedded computers is expected to exceed those from general-purpose systems. Thus, the design process for embedded computer systems is both challenging and important.

The design process for embedded computers differs from that for general-purpose systems. An embedded system designer is less constrained by compatibility considerations, and has more design flexibility. Also, a designer may tune the system for certain applications, because an embedded system is used in very specific settings. On the other hand, the economics often do not justify a custom design because the revenue stream from a particular embedded system design is limited. Therefore, automated design tools are essential to capture the benefits of customization while keeping design costs down.

Automatic design tools take in a design space specification and generate designs that satisfy certain constraints and optimize metrics such as performance, cost, and/or power consumption. Additionally, the user provides a benchmark application together with benchmark inputs. The benchmark is used for generating optimization metrics such as time (inverse performance) estimates. The design space specification concisely represents all designs that meet certain constraints. For instance, a user may be interested in a base processor design and other designs derived from the base processor design where the number of integer units varies in a range from one to four. The user also provides optimization criteria, such as cost or time and suitable evaluation functions for estimating these optimization criteria. For instance, the user may specify that cost of a

design be measured in number of gates. Alternately, the user may require that a chip area metric be used for costing a design.

Consider designing a system where the only optimization criteria are cost and execution time. The cost of the design is evaluated in terms of the physical chip area occupied by the design. The time is evaluated by simulating the design on a specified benchmark application. In this setting, a *Pareto design* is a time-cost optimal design, which is not worse than any other design in the design space, in both time and cost and a *Pareto set* is the set of all Pareto designs. In contrast, quality designs and quality sets are more loosely defined. A *quality design* is not worse in both time and cost to most designs in the design space and a *quality set* is a set of quality designs that span the range from low cost/high time to high cost/low time designs. The objective of an automated design tool is to generate a quality set that approaches a Pareto set. Once an automated design tool generates the quality set, the user can choose the best design that satisfies any cost and time constraints. Additionally, the input to higher-level design tools often consists of quality sets of subsystems. The higher-level tools use these quality sets to explore tradeoffs, e.g. increasing the number of integer units versus increasing the cache size.

Typically, a design is specified by several design parameters, where each design parameter is allowed to take on a specified discrete range of values. A particular design represents a particular value for each of these parameters. For instance, a cache may be specified by parameters such as the cache size, associativity, line size and number of ports. An example of a particular cache design choice is an 8KB two-way set-associative cache with a line size of 32 bytes and one port. The evaluation of a design estimates its optimization metrics. For instance, the optimization metrics of relevance for a cache design are typically its cost in terms of chip area and its impact on performance as measured in number of misses or stall cycles. The cost of a cache design may be estimated quickly and fairly accurately from the cache parameters. Estimating the stall cycles or number of misses requires many hours of computer time, because of the time-consuming nature of processor and cache simulation. Thus, in many situations, the evaluation effort is fairly large in terms of computing resources and more importantly, in terms of actual time.

The size of a design space is the number of distinct designs in the design space. Because the size of a design space is the product of the ranges of each of the design parameters, it increases rapidly with the number of design parameters. An exhaustive exploration of the design space requires evaluating each of the designs in the design space. Because of the computing resources and time required for each evaluation and the sheer number of designs in a typical design space, such exhaustive exploration is usually infeasible. Automatic design space exploration tools must employ efficient methods for exploring the design space by reducing the number of complete designs that are evaluated to a small fraction of the size of the entire design space.

In order to keep the exploration cost manageable, manual designers of computer systems use several approaches. One approach is to decompose the system into components that interact minimally with each other. For instance, a computer system may be decomposed into processor, cache and memory components. Alternative designs for each component are evaluated in isolation to determine the best designs for individual components. Combinations of these designs are then put together to build complete system designs and these system designs are evaluated to determine the best system designs. Since the design space for the individual components is much smaller, and since the number of complete designs that are evaluated is a fraction of the overall design space, the overall design time and effort are reduced substantially. Provided the interactions between components do not elevate suboptimal subsystem designs to optimal system level components, there will be no loss in accuracy. Though there have been attempts to formalize this design process, manual designers generally use *ad hoc* approaches for hierarchical design. This report develops a more rigorous formalized approach for hierarchical design and guarantees the optimality of the final design provided certain conditions are met when the system is decomposed into component subsystems.

In general, component designs may be excluded from further consideration based on validity or quality considerations. A composition puts together component designs and is a valid system only if the component designs mutually satisfy certain system-level validity criteria. Common validity filters at the component level exclude component designs that are determined to be not part of any system design. Partial validity filters eliminate generation of compositions that are not valid

systems. Quality filtering excludes components based on a component-level evaluation that provides optimization metrics. The objective of this report is to reduce the number of system designs that are generated with minimal or no effect on the quality of the final quality set of system designs, through appropriate validity and quality filtering at the component level.

We consider the specific case of an embedded computer system design in detail. In our model, the computer system consists of a hardware accelerator, a general-purpose processor and a memory system. The memory system in turn includes three separate caches. We describe how the theoretical results described in this report can be applied to generate the Pareto set of computer system designs from the individual Pareto sets of processors, hardware accelerator and caches.

2 Pareto set generation using simple composition

In this section, we define the Pareto set more formally, define a decomposition of a system design space into component design spaces and describe a simple composition procedure for generating the system Pareto set.

2.1 Pareto Set for a Design Space

The Pareto set has been developed in the literature in the context of multi-objective or multi-criteria optimization [1-3]. This subsection reviews these concepts and describes the notation that we will use subsequently.

The *design space* D is a set of system designs that are of interest in an automatic design exploration. An evaluation function, $E : D \rightarrow \Re^m$, maps a design, d, to $\mathbf{e} = (e^1,...,e^m)$ in the $m-$dimensional real space, $\Re^m$. Typically, the evaluation function is not an analytic function, and requires time-consuming simulation, especially to assess optimization metrics such as performance.

Consider two designs, d_k and d_l with vector evaluations, $E(d_k)$ and $E(d_l)$ respectively. An evaluation, $E(d_k) \leq E(d_l)$, if each element of $E(d_k)$ is less than or equal to the corresponding element of $E(d_l)$. In this case, d_k is a superior design because the evaluation metrics are defined such that a lower value is more desirable. Similarly an evaluation, $E(d_k) = E(d_l)$, if each element

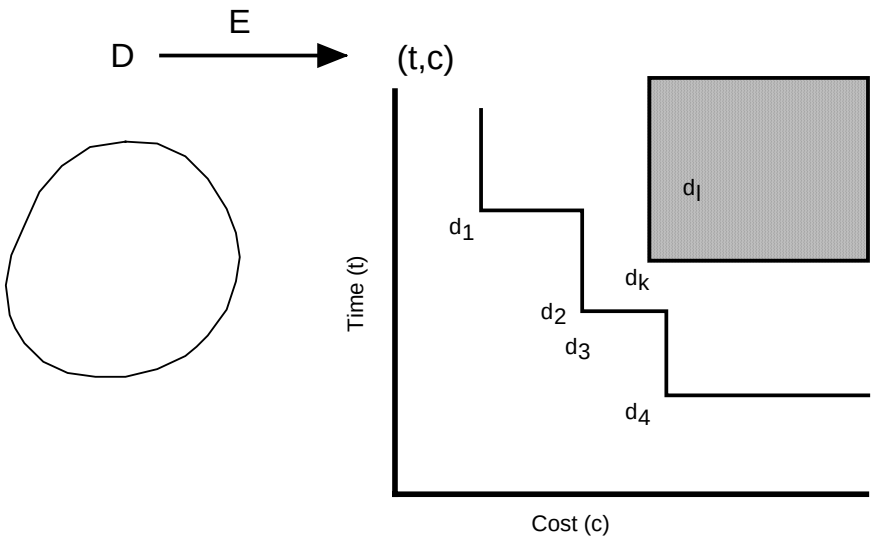

Figure 1: Pareto curve in the two-dimensional time-cost space

of $E(d_k)$ is equal to the corresponding element of $E(d_l)$. A design d_k *eclipses* another design d_l, if $E(d_k) \leq E(d_l)$ and $E(d_k) \neq E(d_l)$. A design, d_k *weakly eclipses* another design d_l if $E(d_k) \leq E(d_l)$. Finally, d_k is *equivalent* to d_l if $E(d_k) = E(d_l)$. In other words, if design d_k is superior to design d_l in at least one of the evaluation metrics and matches d_l in the other evaluation metrics, d_k is superior to design d_l and design d_k eclipses d_l. A design d_k weakly eclipses d_l if it is better than or matches d_l in each of the evaluation metrics. We use the following notation for eclipsing, weakly eclipsing and equivalent respectively: $d_k \prec d_l$, $d_k \preceq d_l$, $d_k \approx d_l$.

In Figure 1, the evaluation function maps all points in the design space, D, to the two-dimensional time-cost plane. In this case, the evaluation function delivers exactly two metrics, time and cost. Design $d_k \prec d_l$ because the cost and time of d_k are both lower than that of d_l. Designs d_2 and d_3 map to the same point in the time-cost plane. Therefore, $d_2 \preceq d_3, d_3 \preceq d_2$, $d_3 \approx d_2$ but $d_2 \not\prec d_3$ and $d_3 \not\prec d_2$.

The *eclipsed region* of d_k is the region in the $m-$dimensional real space, where each of the coordinates is larger than or equal to the corresponding element of the vector, $E(d_k)$. In two-dimensional time-cost space, the eclipsed region of a design, d_k, is the closed quarter plane to the north-east of the point to which d_k maps to as indicated by the shaded region in Figure 1 but does not include the point on which d_k lies. A *Pareto design,* $d_p \in D$, is a design that is not eclipsed by any other design, $d \in D$. A *complete Pareto set,* $P_C(D)$, is the set of all the Pareto designs, $d_p \in D$. In Figure 1, the complete Pareto set is $\{d_1, d_2, d_3, d_4\}$. A *Pareto set,* $P(D)$, is a subset of the complete Pareto set, $P_C(D)$, that includes at least one design from each set of equivalent Pareto designs. In Figure 1, a Pareto set contains $\{d_1, d_4\}$ and one of d_2 or d_3. The *set of Pareto sets,* $\mathcal{P}(D)$, is the set of all the Pareto sets of D. In our example,

$$\mathcal{P}(D) = \langle \{d_1, d_{2,} d_3, d_4\}, \{d_1, d_{2,} d_4\}, \{d_1, d_3, d_4\} \rangle$$

The *eclipsed region of a Pareto set* is the union of all the eclipsed regions of its members. All designs that fall in the eclipsed region of a Pareto set are eclipsed by one or more designs in the Pareto set. The $(m-1)-$dimensional *Pareto surface* (or *Pareto curve* in two-dimensional space) partitions the eclipsed region of a Pareto set from the rest of the $m-$dimensional space. The Pareto designs fall on the Pareto surface between the eclipsed and non-eclipsed regions and there are no designs in the design space that fall strictly within the non-eclipsed region of the Pareto set. Since, in two-dimensional space, the eclipsed region of a Pareto set is the union of quarter planes, the Pareto curve consists of a set of alternating horizontal and vertical segments connecting all the Pareto points together as shown in Figure 1.

2.2 Decomposition

In this subsection, we motivate the benefits of decomposing a system design space into component design spaces and define a decomposition of a system into components. In our framework, a *system* is an electronic device that has a physical realization as one or more integrated

chips, such as an embedded computer system or processor system. A *component* is a device that is part of a system and usually has a physical realization as part of one or more integrated chips. In the context of an embedded computer system, components include processors, instruction or data caches, hardware accelerators. At a finer level of detail, a component may be considered to be a system consisting of smaller components and it may be designed using the methods described in this report.

Consider a system that is subdivided into components or subsystems. Typically, the design of each component is affected only by a small subset of the design parameters. Once the best designs for each component are identified, they are composed together into systems and evaluated. The number of system designs evaluated using this approach is much smaller than the number of system designs in the original design space. The improvement in the efficiency of hierarchical design space exploration is due to three factors.

1. The size of each component's design space is smaller than the original system design space, because the number of relevant design parameters for each component is smaller.

2. The evaluation effort for each component design is low because the component design is less complex.

3. The number of system designs that are evaluated is a small fraction of the system design space.

For instance, consider a design space consisting of 10,000 designs that is partitioned into two design spaces consisting of 100 designs each. Assume that the evaluation of these component designs yields two Pareto sets with 10 designs each. Now, the 100 system designs formed by taking the cross-product of the designs in the two Pareto sets are evaluated to determine the Pareto set for the entire system. A total of 300 designs are evaluated compared to the 10,000 designs in the original design space. Typically, component design evaluation requires significantly less effort than system design evaluation. If component design evaluation effort is insignificant compared to system design evaluation, the total evaluation effort may be reduced by as much as a factor of 100.

Definition: A *decomposition*, $\Theta(D)$, of the design space, D, for an architecture of a system, S, that can be partitioned into components, $C_1, C_2, \cdots, C_n$, consists of:

1. design spaces, $D^i, 1 \le i \le n$, for components, C^i, where $d^i \in D^i$, is a design for C^i. Each component design, $d^i \in D^i$, has a validity vector, $\mathbf{c}^i$, and an evaluation function $E^i : D^i \rightarrow \Re^{m^i}$ that maps a design $d^i \in D^i$ to an $m^i -$ dimensional space.

2. a decomposition function, $\ominus : D \rightarrow \left(D^1 \times D^2 \ldots \times D^n\right)$, maps a system design, $d \in D$, to a set of component designs, $d^i \in D^i$,

3. a validity function, $V : \left(D^1 \times D^2 \ldots \times D^n\right) \rightarrow [TRUE, FALSE]$, maps a set of component designs, $d^i \in D^i$, one from each of the component design spaces, to *TRUE*, indicating that the composition is a valid system, or *FALSE*. The validity function, $V(\ )$, is a Boolean function of the validity vectors, $\mathbf{c}^i$, of the individual component designs.

4. a composition function, $\otimes : \left(D^1 \times D^2 \ldots \times D^n\right) \rightarrow [D, \perp]$, that maps a set of component designs, $d^i \in D^i$, to a system design $d \in D$, iff the validity function, $V(\ )$, returns *TRUE* and otherwise produces bottom, $\perp$, indicating that the composition of the subsystems d^i does not generate a valid system design. The composition of a decomposition of any design, $d \in D$, yields the same design, i.e. $\otimes\left(\ominus\ (d)\ \right) \equiv d, \forall d \in D$.

Thus, the decomposition is one-to-one and the composition is onto. There is a one-to-one relationship between any design and its components, $\left(d^1, \ldots, d^n\right)$, but the space of all combinations of designs may contain an indeterminate number of combinations that map to invalid system designs. Figure 2 shows that each design $d \in D$ is mapped to a set of $d^i \in D^i$ and vice versa. However, some compositions of $d^i \in D^i$ may be mapped to $\perp$.

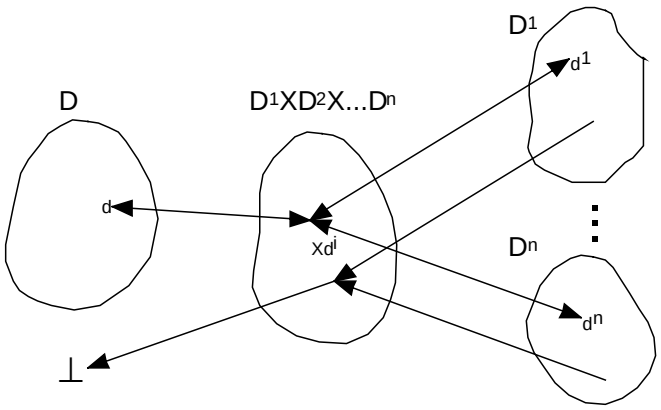

Figure 2: Decomposition of design space

The concepts that apply to the system design space, D, viz., evaluation function, Pareto design, Pareto set, extend to each of the component design spaces D^i. Each design space D^i has an evaluation function $E^i : D^i \rightarrow \Re^{m^i}$ that maps a design $d^i \in D^i$ to an $m^i -$dimensional space. A Pareto design, $d^i_p \in D^i$, is a design for component C^i that is not eclipsed by any other design $d^i \in D^i$. The complete Pareto set $P_C\left(D^i\right)$ of a component design space D^i is the set containing all the Pareto designs in D^i. A Pareto set $P\left(D^i\right)$ of a component design space, D^i may exclude some but not all of each set of equivalent Pareto designs.

In general, a system may have several alternate architectures, where each architecture is associated with its own decomposition. For instance, an embedded system may have two distinct architectures: one consisting of a processor, hardware accelerator and memory and the other consisting of just a processor and a memory. There are three components in the first system architecture and two components in the other. Also, an architecture may have more than one component from a component design space and all these components may not be identical. For instance, a system may have more than one processor from a processor design space and all these processors need not necessarily have the same performance. Thus, the system compositions for each system architecture may be formed by taking the Cartesian product of the component design spaces in its decomposition. The validity function determines that a subset of these compositions are valid.

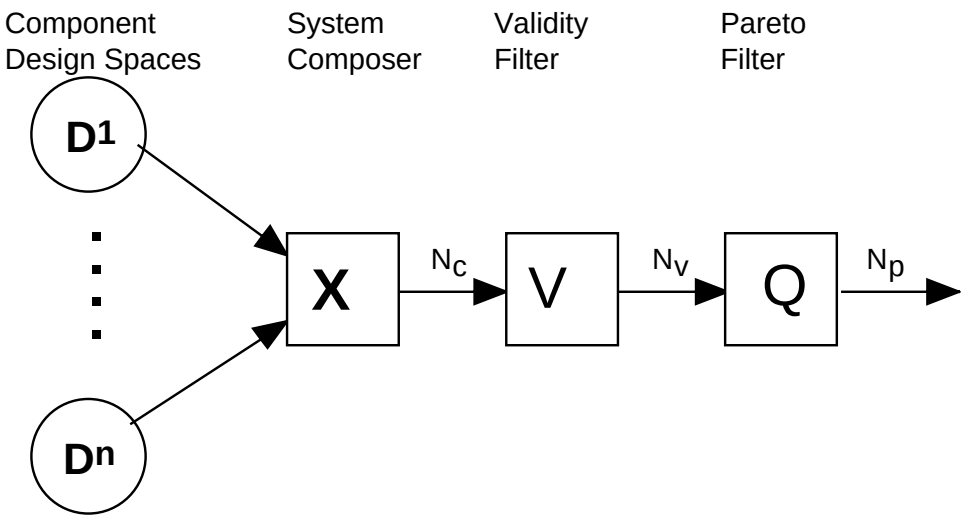

Figure 3: Simple system composition

The entire system design space consists of the union of valid systems of many system architectures. In this report, we consider finding the quality systems for each architecture separately, but there are opportunities for improving the efficiency of design exploration by jointly searching multiple architectures.

2.3 Simple system composition

In this subsection, we describe a simple method for generating a Pareto set of system designs, given a system decomposition and a design space for each of the components.

The left side of Figure 3 shows the component design spaces of a decomposition. The following procedure generates the Pareto set of system designs in a straightforward but inefficient manner. First, we compose all possible combinations of component designs by taking the Cartesian product of component design spaces. Second, we apply the system validity function to each composition and exclude invalid systems. Finally, we evaluate the remaining valid systems and exclude non-Pareto designs as follows. We start with an empty partial Pareto set. Each system design is evaluated and discarded if it is eclipsed by at least one of the current designs in the partial Pareto set. Otherwise, the design is added to the partial Pareto set and any of the current designs that this

design eclipses are removed. Once all system designs are evaluated, we have a Pareto set of system designs.

Let N_c be the number of compositions after the first step, N_v be the number of valid compositions (i.e. system designs) after the second step and N_p be the number of Pareto designs at the end of the process. Consider a system consisting of a processor and a memory component. Assuming that there are 100 designs in each of the processor and memory design spaces, we have a total of 10,000 compositions, i.e. N_c=10,000. Assume that the validity function returns TRUE only for 1,000 of these 10,000 compositions, i.e. N_v=1,000. Finally, assume that after evaluation only 100 of these 1,000 designs are Pareto designs, i.e. N_p=100. Note that only a small fraction of the compositions may eventually turn out to be Pareto designs. We aim to reduce the number of compositions, N_c and the number of systems, N_v so that they are close to N_p, by filtering out (excluding) certain component designs and compositions.

3 Validity and quality filtering

In this section, we describe the overall procedure for validity and quality filtering of the component design spaces. First, we describe how the system validity function is expanded. We describe how the transformed system validity function is used to derive component-level validity functions that can be used for filtering out component designs and generating a set of predicated validity sets for each component. Finally, we apply a quality filter to each predicated validated set.

3.1 Expansion of validity function

The system validity function, $V()$, specifies whether a composition forms a valid system. The validity function satisfies the following properties. First, we assume that the validity function is expressible in canonical Boolean OR-AND form, i.e. the Boolean validity function consists of an OR of several expressions and each of these expressions consists of an AND of several terms. A *term* in an AND expression is the smallest component that evaluates to Boolean values, i.e., *TRUE* or *FALSE*. Second, we assume that the terms contain Boolean functions of elements of the validity vector, $\mathbf{c}^i$, associated with a particular component design. Finally, we assume that parameters can assume a small discrete range of values. This assumption guarantees that the expansion procedure

described subsequently succeeds and does not produce an unwieldy expansion of the system validity function.

Since any Boolean function can be reduced to canonical OR-AND form, these assumptions are not unduly restrictive. However, we cannot handle some decompositions where the validity of a composition is an algorithmic function of the validity parameters, i.e. determining the validity of a composition requires executing a program with the validity parameters as input. Even in these cases our methods may be partially effective, if we can partition the validity function to the AND of two terms, one of which requires an algorithmic evaluation and the other satisfies the requirements stated above. We discuss this further in a later section.

As an example, consider a system consisting of a processor and a memory. Let *instrSize*, *intLitSze* and *memLitSize* be parameters of the processor corresponding to the instruction size, integer literal size and memory literal size. Additionally, let *procPorts* be the number of data access ports of the processor and *memPorts* be the number of ports provided by the memory. These parameters may or may not be the ones used in specifying an enumeration of the component design space. For example, the specification of the processor and memory design space may include ranges for *procPorts* and *memPorts* respectively. But, *instrSize* is perhaps derived from evaluating a processor design, specified using some set of other parameters. Let the system validity function be

$$V() = \begin{bmatrix} ((instrSize \le 64) \wedge (procPorts \le memPorts) \wedge (intLitSize \le 32)) \vee \\ ((instrSize \le 64) \wedge (procPorts \equiv memPorts) \wedge (memLitSize \le 32)) \end{bmatrix}$$

The validity function is the OR of two AND expressions:

$$((instrSize \le 64) \wedge (procPorts \le memPorts) \wedge (intLitSize \le 32))$$

and

$$((instrSize \le 64) \wedge (procPorts \equiv memPorts) \wedge (memLitSize \le 32))$$

The terms in this validity function are:

$(instrSize \le 64)$,

$(procPorts \le memPorts)$,

$(intLitSize \leq 32)$,

$(procPorts \equiv memPorts)$ and

$(memLitSize \leq 32)$.

A *common term* is a term that occurs in all AND expressions and involves parameters that are all associated with a particular component. All other terms are classified as *partial terms*. The component parameters that appear in these partial terms are *partial parameters*. All other parameters are *common parameters*. The term $(instrSize \leq 64)$ appears in both AND expressions and is therefore a common term. The other four terms are partial terms. The parameters, *procPorts*, *memPorts*, *intLitSize* and *memLitSize*, appearing in the partial terms are partial parameters. The parameter *instrSize* is a common parameter.

Since the common terms appear in all AND expressions of the system validity function, they can be factored out, so that the system validity function is the AND of a CE (common expression) and a PSOP (partial sum of products). The CE is the AND of all common terms and the PSOP is the original system validity function with the common terms eliminated in each AND expression. Thus,

$$V() = CE \wedge PSOP \qquad (1)$$

In our example,

$CE = (instrSize \leq 64)$ and

$$PSOP = \begin{bmatrix} ((procPorts \leq memPorts) \wedge (intLitSize \leq 32)) \vee \\ ((procPorts \equiv memPorts) \wedge (memLitSize \leq 32)) \end{bmatrix}$$

The PSOP contains two types of terms. A *singleton term* is a partial term that contains parameters associated with exactly one component. A *coupled term* is a partial term that contains parameters associated with two or more components.

In our example, $(intLitSize \leq 32)$ and $(instrSize \leq 64)$ are singleton terms because the parameters are associated with the processor and $(procPorts \leq memPorts)$ and

$(procPorts \equiv memPorts)$ are coupled terms because each term involves one parameter associated with the processor and another associated with the memory.

We now describe the expansion procedure that is performed on the PSOP to convert all coupled terms into singleton terms. We iterate over all AND expressions in the PSOP and secondarily, over all coupled terms in each AND expression. For each coupled term, we choose one parameter as the expansion parameter. We expand this parameter by iterating through all the values that the expansion parameter can take for the designs in the associated component design space. We replace the original AND expression by an OR of several AND expressions, each corresponding to a particular value for the expanded parameter. Each AND expression is the conjunction of two terms. One term asserts that the expansion parameter has a certain value and the other term is the original AND expression with the expansion parameter set to the same value.

In our previous example, $(procPorts \leq memPorts)$ is a coupled partial parameter. Choose *procPorts* as an expansion parameter. Assume that the design space consists of processors with one or two data access ports. Substituting *procPorts*=1 and *procPorts*=2 in the validity function, we obtain an equivalent validity function which does not contain any coupled partial terms:

$$
\begin{aligned}
&PSOP = \\
&((procPorts \equiv 1) \wedge (memPorts \geq 1) \wedge (intLitSize \leq 32)) \vee \\
&((procPorts \equiv 2) \wedge (memPorts \geq 2) \wedge (intLitSize \leq 32)) \vee \\
&((procPorts \equiv 1) \wedge (memPorts \equiv 1) \wedge (memLitSize \leq 32)) \vee \\
&((procPorts \equiv 2) \wedge (memPorts \equiv 2) \wedge (memLitSize \leq 32))
\end{aligned}
$$

The expansion scheme outlined above places a set of equality constraints on the coupled partial parameter being expanded. In general, we may choose any set of constraints on the expansion parameter provided the following holds. The set of constraints must be such that every value that this parameter can take for designs in the component design space satisfies at least one of the constraints. For instance, we could have chosen to expand *procPorts* in $(procPorts \leq memPorts)$ as $procPorts \leq 1$ and $procPorts \geq 2$. Ideally, we choose the constraints on the expansion parameter to reduce or minimize the number of terms in the expanded validity function.

After the expansion is complete, there are no more coupled terms in the expanded system validity function. We now use the commutativity of terms in an AND expression to group together all terms with parameters associated with a particular component. The system validity function is now in the form:

$$\begin{aligned} V() &= \left[\bigwedge_{i=1}^{n} CE^i \right] \wedge \left[\bigvee_{j=1}^{m} PE_j \right] \\ &= \left[\bigwedge_{i=1}^{n} CE^i \right] \wedge \left[\bigvee_{j=1}^{m} \left(\bigwedge_{i=1}^{n} CPE_j^i \right) \right] \end{aligned} \tag{2}$$

where CE^i consists of the part of the common expression containing the terms associated with component, C^i, PE_j is the jth AND expression in the expanded PSOP, CPE_j^i consists of the part of PE_j containing the terms associated with component, C^i, n is the number of components composing a system and m is the number of AND expressions in the expanded PSOP.

Assuming that the processor is the first component and memory the second component, we have:

$$\begin{aligned} &CE^1 = (instrSize \leq 64) \qquad CE^2 = TRUE \\ &CPE_1^1 = (procPorts \equiv 1) \wedge (intLitSize \leq 32) \\ &CPE_1^2 = (memPorts \geq 1) \\ &CPE_2^1 = (procPorts \equiv 2) \wedge (intLitSize \leq 32) \\ &CPE_2^2 = (memPorts \geq 2) \\ &CPE_3^1 = (procPorts \equiv 1) \wedge (memLitSize \leq 32) \\ &CPE_3^2 = (memPorts \equiv 1) \\ &CPE_4^1 = (procPorts \equiv 2) \wedge (memLitSize \leq 32) \\ &CPE_4^2 = (memPorts \equiv 2) \end{aligned} \tag{3}$$

In this section, we described how to generate an expanded system validity function. As illustrated above, an expanded system validity function is composed of a set of expressions. Each expression is a conjunction of singleton terms that are associated with a particular component. In the next section,

we utilize the structure of the expanded system validity function to construct common and partial validity filters

3.2 Component-level Filtering

Figure 4 illustrates an efficient procedure for generating the system Pareto set through filtering component designs. As in Figure 3, the left-hand side inputs consist of the component design spaces and the outputs are a set of quality system designs. The additional levels of validity and quality filtering at the component level reduce the number of compositions, N_c and number of valid systems, N_v that are generated at the system level compared to the simple composition algorithm described in Section 2.3. Ideally, $N_c = N_v = N_p$. Recall that the system design space is much larger than individual component design spaces and that system evaluation is much more expensive than component evaluation. Thus, our component-level filtering procedure aims to reduce the number of system designs that are evaluated and filtered out.

The rest of this section describes the different levels of filtering in Figure 4 from left to right. Each level of component filtering can be applied independently of the other levels to its right. We illustrate the effectiveness of a particular degree of filtering using our running example consisting of processor and memory components.

3.3 Common validity filtering

We now describe the first level of filtering labeled "Common Validity Filter" in Figure 4. The objective of common validity filtering is to remove component designs that are not part of any valid system from further consideration.

Consider a design, $d^i \in D^i$ that is part of the component design space for component, C^i. Since the parameters of the predicate, CE^i, are associated with C^i, we can evaluate CE^i for a particular design d^i. The expanded system validity function is the conjunction of the predicate CE^i and other terms. Therefore, if the predicate CE^i is *FALSE*, for a design d^i, the expanded system validity function is also *FALSE* for any composition containing d^i. Therefore, any design d^i that does not

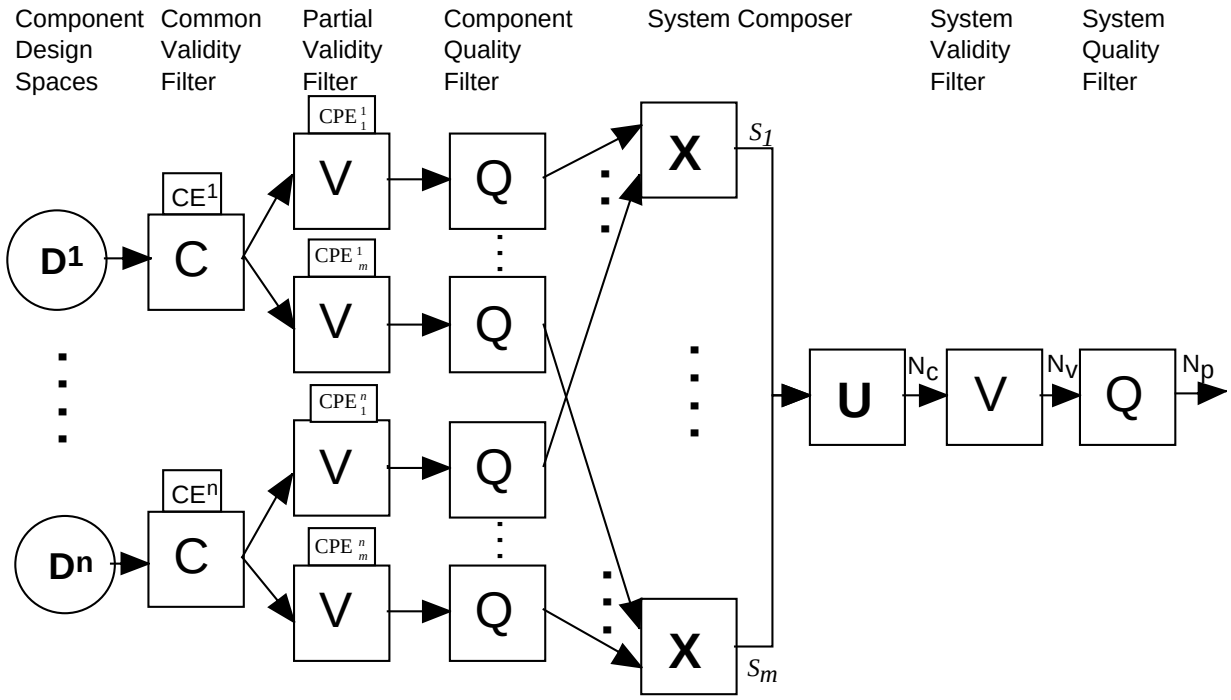

Figure 4: Interface Validity Filtering at Component Level

satisfy the predicate CE^i can be eliminated from further consideration, without affecting the set of valid systems that are eventually generated. This statement forms the basis for common validity filtering.

In Figure 4, the column of boxes labeled with the letter C shows the common validity filters. The small boxes labeled CE^1 through CE^n indicate the predicates that are used for filtering. A common validity filter for component C^i accepts all the designs in D^i as input and outputs only those designs that satisfy the predicate CE^i.

In our running example, consisting of a processor and memory design space each containing 100 designs, assume that the common term $(instrSize \leq 64)$ is satisfied by only 40 of the 100 processor designs. Assuming that the remaining component-level filters are not applied, the number of system

compositions, N_c, generated by the composition step is reduced to 4,000, compared to 10,000 without common validity filtering.

3.4 Partial validity filtering

We describe the second level of filtering labeled "Partial Validity Filter" in Figure 4. The objective of partial validity filtering is to form a set of predicated validity sets for each component. A composition picks component designs from identically numbered predicated validity sets to generate valid systems.

Let PS_j^i denote the *j*th predicated validity set for the *i*th component, consisting of the designs in the design space D^i that satisfy the component partial expression, CPE_j^i and the common expression, CE^i. In Figure 4, the column of boxes labeled with the letter, V, show the partial validity filters. The small boxes labeled CPE_1^1, CPE_m^1, CPE_1^n, CPE_1^n above these indicate the predicates that are used for generating the predicated validity sets, PS_1^1, PS_m^1, PS_m^1, PS_m^1 respectively. Assume for the rest of this subsection (3.4), that the filters labeled Q are absent. The column of boxes labeled X, show the system composers. There are a total of *m* system composers, but we only show the first and the last composers. The *j*th composer accepts the *j*th predicated validity set from each of the components and generates system compositions by taking the Cartesian product of all these *n* validity sets. The box labeled U forms the union of the system compositions generated by the *m* system composers. We show later that these system compositions are exactly the set of all valid systems that can be generated from the component design spaces.

In our example, the input to the partial validity filters for the processor design space consists of the 40 processor designs that satisfy $(instrSize \leq 64)$. The partial validity filters generate four predicated validity sets, each satisfying one of the four component partial predicates. Assuming that a processor design may have both an $(intLitSize \leq 32)$ and a $(memLitSize \leq 32)$, these validity sets are not disjoint and a design may belong to two (or more, in general) predicated validity sets. Assume that partial validity filtering generates 12 processor designs and 50 memory designs in each

predicated validity set. Then, the number of compositions generated is 4*12*50= 2,400. There are only 1,000 valid system designs and therefore many of these compositions are identical. The system composer box labeled U filters out these identical compositions and generates the 1,000 valid systems. The final Pareto filtering yields 100 Pareto designs. Thus, $N_c = 1{,}000$, $N_v = 1{,}000$ and $N_p = 100$. Partial validity filtering attempts to ensure that only those system compositions that are valid are generated.

3.5 Quality filtering

We describe the third and final level of component filtering labeled "Component Quality Filter" in Figure 4. As described earlier, each component design space and the system design space have evaluation functions, which generate real-valued optimization vectors. A component-level evaluation function is *correlated* if superior component designs as measured by the component optimization metric usually generate superior system designs. Specifically, a component-level evaluation is *perfectly correlated* if the substitution of a superior component-level design in a system design always yields a superior system design as measured by system-level optimization metrics. In Section 4.3, we relate the notion of correlation to monotonic evaluation functions.

In our example, the optimization metrics of a processor design consists of cost in chip area and execution time in cycles on a specified benchmark. The optimization metrics of a memory design consist of cost in chip area and its impact on execution time in stall cycles. The system evaluation consists of chip area and execution time, where each metric is the sum of the corresponding processor and memory metrics. In this case, the processor and memory evaluations are perfectly correlated. Substituting a memory design with smaller chip area and stall cycles contribution always generates a better system design.

In Figure 4, the column of boxes labeled with the letter Q shows the quality filters. We show only the first and last quality filters for the first and last components, but there are a total of *m.n* quality filters. A quality filter evaluates designs and filters out those designs that are likely to be worse than other designs, i.e. likely to be eclipsed. A Pareto filter is a quality filter that retains only those designs that are not eclipsed by any other design. The *j*th composer accepts the *j*th predicated

quality set from each of the components and generates system compositions by taking the Cartesian product of all these *n* quality sets. The box labeled U forms the union of the system compositions generated by the *m* system composers.

Consider evaluating all the component designs in PS_j^i and retaining only the quality (Pareto) set, QS_j^i, of designs that are not eclipsed by any other design. We can always improve a system design containing a component design *d* in PS_j^i but not in QS_j^i, by substituting a design d_p in QS_j^i that eclipses *d*. Since *d* and d_p, belong to the same predicated validity set, the substitution is guaranteed to yield a valid system. When the component evaluation functions are correlated, quality filtering can reduce the number of system compositions generated with minimal impact on the final system Pareto set. In Section 4.3, we show that component quality filtering does not affect the quality of the final system Pareto, provided the component evaluation functions are perfectly correlated.

In our example, each processor design is evaluated to estimate its execution time in cycles and its cost in chip area. Also, each memory design is evaluated to estimate its contribution to stall cycles and its cost in chip area. We may choose to include or substitute these metrics with other metrics such as power consumption. Assume that quality filtering reduces the number of processor designs from 12 to five and the number of memory designs from 50 to 10 for each of the four predicated validity sets. Then, the total number of systems generated by the system composer is reduced to 4*5*10=200. Thus, $N_c = 200$, $N_v = 200$ and $N_p = 100$. Quality filtering at the component level greatly reduces the number of system compositions and valid systems that have to be evaluated at the system level.

4 Characterization of filtering

In this section, we characterize the effectiveness of filtering. First, we show that the validity filtering procedure does not filter out any valid system design. We then discuss the concept of monotonicity as it applies to decompositions and their component and system evaluations. We show that our procedure generates the system Pareto set, provided certain monotonicity requirements are satisfied. If the component-level evaluations are approximate but their inaccuracy is bounded, we

can still obtain the system Pareto set under certain conditions. We provide faster methods for generating the system quality set, some of which generate the Pareto set and others an approximation to the Pareto set. Finally, when the interactions between subsystems are too complex to be handled by any our results, the process of combining subsystem Pareto curves may not generate a system Pareto set but may be a useful approximation.

4.1 Characterization of validity filtering

In the previous section, we described two levels of validity filtering. Common validity filtering eliminates component-level designs that are not part of any valid system and partial validity filtering forms a number of predicated validity sets. We show that the validity filtering has no effect on the set of valid system designs that are generated.

Lemma 1: The system design space and the set of systems generated by the validity filtering procedure are identical, i.e.,

$$D = \bigcup_{j=1}^{m} \bigotimes_{i=1}^{n} \left(PS_j^i \right) \tag{4}$$

Proof: We first show that any design in the union of compositions is in the system design space, *D*. Consider a composition formed by picking a set of component designs, $d^i \in PS_j^i$ from the *j*th predicated validity set of each of the components. Since each design, $d^i \in D^i$ satisfies its common expression, CE^i, this composition satisfies the conjunction of all the common expressions. Since each design, d^i satisfies the *j*th component partial expression, CPE_j^i, the composition also satisfies the partial expression, PE_j. Therefore the composition satisfies the expanded system validity function and is a valid system. Thus, valid system designs may be generated by taking the Cartesian product of identically numbered predicated validity sets.

Consider a design, *d*, in the system design space, *D*, which, by definition, satisfies the system validity function. From Equation (2), it must satisfy all the common expressions, CE^i, and at least one of the partial expressions, PE^j. Therefore, the component designs, d^i, must each satisfy its

common expression, CE^i, and jointly satisfy, say the kth, partial expression, PE_k. Each component design d^i belongs to the kth partial validity set, PS_k^i. Therefore, the composition procedure will compose the designs d^i one from each of the n component design spaces to form d.

First we showed that $D \supseteq \bigcup_{j=1}^{m} \bigotimes_{i=1}^{n} \left(PS_j^i\right)$ and then showed that $D \subseteq \bigcup_{j=1}^{m} \bigotimes_{i=1}^{n} \left(PS_j^i\right)$, Therefore, $D = \bigcup_{j=1}^{m} \bigotimes_{i=1}^{n} \left(PS_j^i\right)$. ■

4.2 Strong and weak monotonic decompositions

In the remaining subsections of Section 4, we characterize the quality filtering step of the overall system composition procedure of Section 3. In this subsection, we characterize decompositions and their associated evaluation functions as weak or strong monotonic. In later subsections, we characterize the system quality sets generated by the overall system composition procedure.

In the context of a decomposition, Θ, of D, a design, d_k, *componentwise weakly eclipses* another design, d_l, if each of the component designs d_k^i in $\oplus(d_k) = \left(d_k^1 \ldots d_k^n\right)$ weakly eclipses the corresponding component design d_l^i in $\oplus(d_l) = \left(d_l^1 \ldots d_l^n\right)$.

Definition: A decomposition, Θ, of D, together with the evaluation functions E and $E^i, 1 \le i \le n$ is *weak monotonic* if for every two designs, $d_k, d_l \in D$,

1. if d_k componentwise weakly eclipses d_l, then d_k weakly eclipses d_l and
2. if $d_k^i \approx d_l^i, \forall i$, then $d_k \approx d_l$.

In contrast, d_k *componentwise strongly eclipses* d_l, if each of the component designs d_k^i in $\oplus(d_l) = \left(d_l^1 \ldots d_l^n\right)$ weakly eclipses the corresponding component design d_l^i in $\oplus(d_k) = \left(d_k^1 \ldots d_k^n\right)$ and at least one component design d_k^i eclipses d_l^i.

Definition: A decomposition, Θ, of D, together with the evaluation functions E and $E^i, 1 \le i \le n$ is *strong monotonic* if for every two designs, $d_k, d_l \in D$,

1. if d_k componentwise strongly eclipses d_l, then d_k eclipses d_l and

2. if $d_k^i \approx d_l^i, \forall i$, then $d_k \approx d_l$.

As is shown in the following Lemma 2, strong monotonicity implies weak montonocity,.

Lemma 2: A decomposition, Θ, of D, that is strong monotonic is also weak monotonic.

Proof: Consider a strong monotonic decomposition, Θ, of D. Any two designs, $d_k, d_l \in D$, decompose into component designs, $\oplus(d_k) = (d_k^1 \ldots d_k^n)$ and $\oplus(d_l) = (d_l^1 \ldots d_l^n)$ respectively. Let d_k componentwise weakly eclipse d_l. First, consider the case where the corresponding evaluations of the component designs are exactly equal, i.e. $d_k^i \approx d_l^i$. By the definition of strong monotonicity, $d_k \approx d_l$. Next, consider the case where at least one component design d_k^i strongly eclipses d_l^i. Again, by the definition of strong monotonicity, $d_k \prec d_l$. Both conditions for weak monotonicity are satisfied. ■

Consider the case where the evaluation functions map a design to the two-dimensional time-cost plane. The evaluation function, $\mathrm{E}^i(d^i)$, generates a time metric, $t(d^i)$ and cost metric, $c(d^i)$ for each design $d^i \in D^i$. The system evaluation function, $\mathrm{E}(d)$, generates a time metric, $t(d)$, and cost metric, $c(d)$ for each design $d \in D$. Let $t(d) = f^t(t(d^i))$, where $f^t(\ )$ is a function that takes the n arguments $t(d^i)$ and generates the time metric for the system design. Similarly, let $c(d) = f^c(c(d^i))$. A real-valued function, $f(\mathbf{x})$, of the vector, $\mathbf{x}$, is *monotonically non-decreasing* if $\mathbf{x} \ge \mathbf{y} \Rightarrow f(\mathbf{x}) \ge f(\mathbf{y})$ and is *monotonically increasing* if $\mathbf{x} > \mathbf{y} \Rightarrow f(\mathbf{x}) > f(\mathbf{y})$.

Lemma 3: If a decomposition has monotonically non-decreasing evaluation functions that map the time and cost of the component designs to the time and cost of the system design respectively,

then the decomposition is weak monotonic. Similarly, if a decomposition has monotonically increasing evaluation functions that map the time and cost of the component designs to the time and cost of the system design respectively, then the decomposition is strong monotonic.

Proof: Consider designs, $d_k, d_l \in D$, $\oplus(d_k) = (d_k^1 \ldots d_k^n)$, $\oplus(d_l) = (d_l^1 \ldots d_l^n)$. Let $\mathbf{t_k} = (t(d_k^1), t(d_k^2), \ldots, t(d_k^n))$ and $\mathbf{t}_l = (t(d_l^1), t(d_l^2), \ldots, t(d_l^n))$.

First, consider the case where $f^t(\,)$ and $f^c(\,)$ are monotonically non-decreasing functions. Assume $d_k^i \preceq d_l^i$, $1 \le i \le n$. By assumption, $\mathbf{t_k} \le \mathbf{t_l}$. By assumption and definition, $f^t(\mathbf{t_k}) \le f^t(\mathbf{t_l})$, which implies that $t(d_k) \le t(d_l)$. Similarly, $c(d_k) \le c(d_l)$. Therefore, $d_k \preceq d_l$ and the decomposition is weak monotonic.

Consider the second case where $f^t(\,)$ or $f^c(\,)$ are monotonically increasing functions. Assume $d_k^i \preceq d_l^i$, $1 \le i \le n$ and that at least one $d_k^i \prec d_l^i$. Without loss of generality, assume the time component of one of the d_k^i is less than that of d_l^i. By assumption, $\mathbf{t_k} < \mathbf{t_l}$. By assumption and definition, $f^t(\mathbf{t_k}) < f^t(\mathbf{t_l})$, which implies that $t(d_k) < t(d_l)$. As in the earlier case, $c(d_k) \le c(d_l)$. Therefore, $d_k \prec d_l$ and the decomposition is strong monotonic. ■

4.3 Exact system Pareto set from component Pareto set

In this section, we show that we can derive an exact system Pareto set from a suitable set of subsystem Pareto sets provided that the decompositions are monotonic. If the decompositions satisfy weak monotonicity, we obtain a system Pareto. If they satisfy strong monotonicity, we obtain the complete system Pareto. In the overall system composition procedure in Figure 4, we use the Pareto filter for each of the quality filters for the predicated component design spaces. A Pareto filter applied to a predicated validity set retains only designs that are not eclipsed by any other design in the predicated validity set. We begin by extending the concept of eclipsing to design spaces, as opposed to individual designs.

Definition: Given two sets of designs, D_1 and D_2, D_1 *weakly eclipses* D_2, ($D_1 \preceq D_2$) if for all $d_2 \in D_2$, $\exists\, d_1 \in D_1$, such that $d_1 \preceq d_2$.

Lemma 4: If $D_1 \preceq D_2$, then $P(D_1) \preceq P(D_2)$.

Proof: Let $d_2 \in P(D_2)$. Since $P(D_2) \subseteq D_2$, $d_2 \in D_2$. Because $D_1 \preceq D_2$, there exists $d_1 \in D_1$ that weakly eclipses d_2. Either this $d_1 \in P(D_1)$ or there is some other $d_3 \in P(D_1)$ that eclipses d_1. So, for every $d_2 \in P(D_2)$, there is a $d \in P(D_1)$ that weakly eclipses it. Therefore, $P(D_1)$ weakly eclipses $P(D_2)$. ■

Lemma 5: If $D_1 \subseteq D_2$ and $D_1 \preceq D_2$, then a Pareto set of D_1, $P(D_1)$, is a Pareto set of D_2.

Proof: Note that we aim to prove that $P(D_1)$ is a Pareto set of D_2, not necessarily the complete Pareto set, $P_C(D_2)$. Our proof strategy is to show that each $d_1 \in P(D_1)$ belongs to $P_C(D_2)$ and every $d_2 \in P_C(D_2)$ either belongs to $P(D_1)$ or has identical evaluation metrics as some $d_1 \in P(D_1)$.

Let $d_1 \in P(D_1)$. Because $D_1 \subseteq D_2$, $d_1 \in D_2$. Assume $d_1 \notin P_C(D_2)$. Then, some $d_2 \in P_C(D_2)$ must eclipse d_1. Since $D_1 \preceq D_2$, $\exists\, d_3 \in D_1$, $d_3 \neq d_1, d_3 \preceq d_2$. By transitivity, $d_3 \prec d_1$. Therefore, $d_1 \notin P(D_1)$, a contradiction. So, $d_1 \in P_C(D_2)$, and $P(D_1) \subseteq P_C(D_2)$.

Let $d_2 \in P_C(D_2)$. Assume $d_2 \notin P(D_1)$. Since $D_1 \preceq D_2$, there must be some $d_1 \in P(D_1)$ that weakly eclipses d_2. If this d_1 does not have the same evaluation metrics as d_2, $d_1 \prec d_2$. Since $D_1 \subseteq D_2$, $d_1 \in D_2$ and d_2 is not a Pareto design in D_2, a contradiction. Therefore, $d_2 \in P_C(D_1)$ and $P_C(D_2) - P(D_1)$ only consists of designs with evaluation metrics that are already represented in $P(D_1)$. Therefore, $P(D_1)$ is a Pareto set of D_2. ■

Figure 5 illustrates Lemma 5. As usual, the left hand side of the figure shows the design spaces, with $D_1 \subseteq D_2$. The right hand side shows the time-cost evaluation plane. The Pareto curve runs

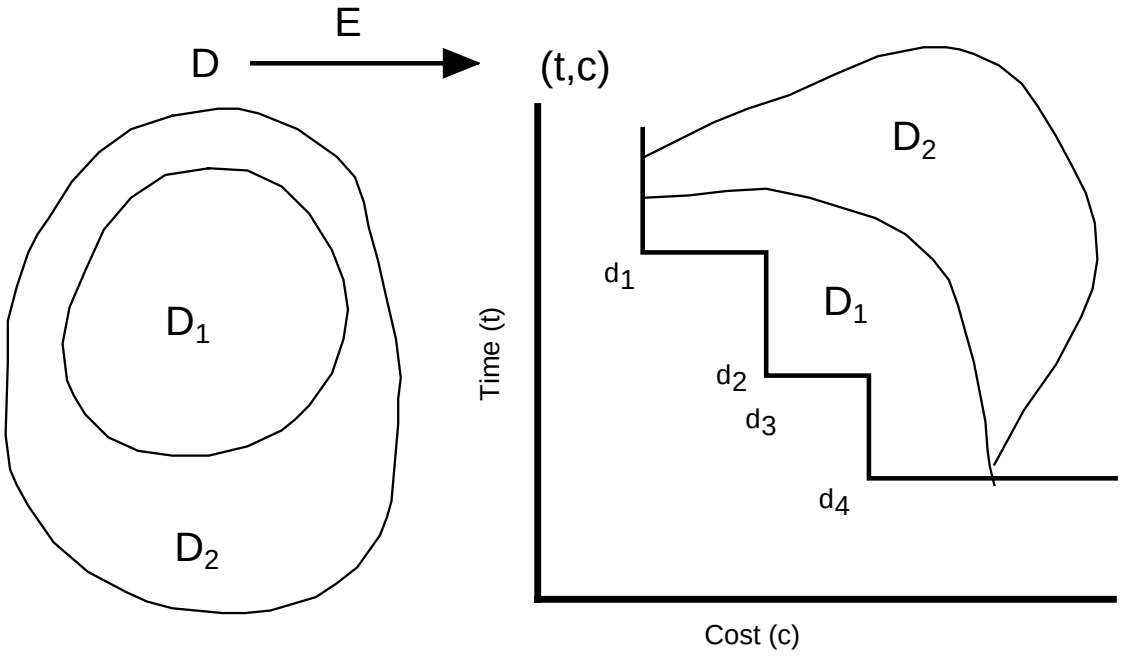

Figure 5: Relation between eclipsing design spaces

through the time-cost mappings of the designs in the complete Pareto set of D_2, $P_C(D_2) = \{d_1, d_2, d_3, d_4\}$. As in our previous examples, d_2 and d_3 map to the same point in the time-cost plane. The design spaces D_1 and D_2 map to the time-cost regions labeled D_1 and D_2 respectively, Since $D_1 \subseteq D_2$, the designs in D_1 map to a smaller subset region compared to D_2. Lemma 5 states that if $D_1 \preceq D_2$, a Pareto set of D_1, $P(D_1)$, must include the designs, that constitute a Pareto set, $P(D_2)$, of D_2, viz. d_1, d_4 and one of d_2 or d_3.

Theorem 1:

In a weak monotonic decomposition, $\Theta(D)$, the overall composition procedure where a Pareto filter is used for each of the quality filters at the component and system level generates a Pareto set of the systems, i.e.,

$$P\left(\bigcup_{j=1}^{m}\bigotimes_{i=1}^{n}\left(P\left(PS_j^i\right)\right)\right)\in \mathcal{P}(D) \qquad (5)$$

Proof: Consider a design, $d \in D$. According to Lemma 1, for each such d there is a corresponding set of component designs, $d^i \in PS_k^i, 1 \le i \le n$, for some value of k, such that their composition, $\bigotimes_{i=1}^{n}\left(d^i\right)$ generates d. For each such $d^i \in PS_k^i$, $\exists d_p^i \in P\left(PS_k^i\right)$, such that $d_p^i \preceq d^i$. By Lemma 1, the composition, $\bigotimes_{i=1}^{n}\left(d_p^i\right)$ is a design, $d_p \in D$. The weak monotonicity assumption guarantees that $d_p \preceq d$. Therefore, $\bigcup_{j=1}^{m}\bigotimes_{i=1}^{n}\left(P\left(PS_j^i\right)\right) \preceq D$. By Lemma 1, any design in $\bigcup_{j=1}^{m}\bigotimes_{i=1}^{n}\left(P\left(PS_j^i\right)\right)$ belongs to D. Therefore, by Lemma 5, $P\left(\bigcup_{j=1}^{m}\bigotimes_{i=1}^{n}\left(P\left(PS_j^i\right)\right)\right)$ is a Pareto set of D. ■

Figure 6 illustrates the proof strategy in Theorem 1. A design, $d \in D$, decomposes into $d^1 \in PS_k^1$ and $d^n \in PS_k^n$. There must be Pareto designs, $d_p^1 \in P\left(PS_k^1\right)$ and $d_p^n \in P\left(PS_k^n\right)$, such that $d_p^1 \preceq d^1$ and $d_p^n \preceq d^n$. These Pareto designs of subsystems must compose into a system design, $d_p \in D$. This design, d_p, which may or may not be a system Pareto design, must weakly eclipse d, because the decomposition is weak monotonic.

A weakness of the result in Theorem 1 is that the composition procedure generates a Pareto set, not necessarily the complete Pareto set. We now show that under strong monotonicity, the composition procedure does generate the complete system Pareto set from the complete Pareto sets of the components. Recall that under weak monotonicity, a system design, d_k, weakly eclipses another system design, d_l, if d_k componentwise weakly eclipses d_l. In contrast, under strong monotonicity, d_k eclipses d_l, if d_k componentwise strongly eclipses d_l. As stated in Lemma 3, strong monotonicity is associated with monotonically increasing system-level evaluation functions whereas weak monotonicity is associated with monotonically non-decreasing evaluation functions.

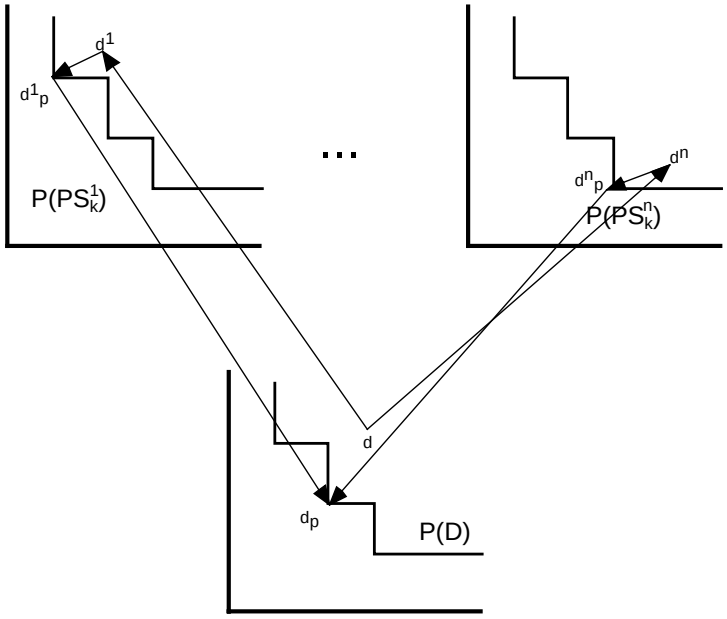

Figure 6: Decomposition of system design and composition of component Pareto designs

Though this result was derived for the specific case of two system-level evaluation functions generating two evaluation metrics, these results are perhaps valid more generally. Since strong monotonicity is a stronger assumption, we are able to strengthen the results in Theorem 1.

Theorem 2:

In a decomposition, $\Theta(D)$, that is strong monotonic, the overall composition procedure where a complete Pareto filter is used for each of the quality filters at the component and system level generates a complete Pareto set of the systesm, i.e.,

$$P_C(D) = P_C\left(\bigcup_{j=1}^{m} \bigotimes_{i=1}^{n} \left(P_C\left(PS_j^i\right)\right)\right) \tag{6}$$

Proof: Consider a design, $d \in P_C(D)$. According to Lemma 1, for each such *d* there is a corresponding set of component designs, $d^i \in PS_k^i, 1 \le i \le n$, for some value of *k*, such that their composition, $\bigotimes_{i=1}^{n}\left(d^i\right)$generates d. For each such $d^i \in PS_k^i$, $\exists d_p^i \in P_C\left(PS_k^i\right)$, such that $d_p^i \preceq d^i$. By

Lemma 1, the composition, $\bigotimes_{i=1}^{n}\left(d_p^i\right)$ is a design, $d_p \in D$. Assume at least one $d_p^i \prec d^i$. Then, design d_p componentwise strongly eclipses d. Because the decomposition is strong monotonic, $d_p \prec d$ and d is not a system Pareto design, a contradiction. Therefore, $d_p^i \approx d^i, \forall i$. Then, since $d_p^i \in P_C\left(PS_k^i\right), \quad d^i \in P_C\left(PS_k^i\right), \forall i$ and $d \in \bigcup_{j=1}^{m} \bigotimes_{i=1}^{n}\left(P_C\left(PS_j^i\right)\right)$. Because $d \in P_C(D)$ and $d \in \bigcup_{j=1}^{m} \bigotimes_{i=1}^{n}\left(P_C\left(PS_j^i\right)\right)$, $d \in P_C\left(\bigcup_{j=1}^{m} \bigotimes_{i=1}^{n}\left(P_C\left(PS_j^i\right)\right)\right)$. Thus, $P_C(D) \subseteq P_C\left(\bigcup_{j=1}^{m} \bigotimes_{i=1}^{n}\left(P_C\left(PS_j^i\right)\right)\right)$.

Assume $P_C(D) \neq P_C\left(\bigcup_{j=1}^{m} \bigotimes_{i=1}^{n}\left(P_C\left(PS_j^i\right)\right)\right)$. Then, $\exists d \in D$, such that $d \in P_C\left(\bigcup_{j=1}^{m} \bigotimes_{i=1}^{n}\left(P_C\left(PS_j^i\right)\right)\right)$ and $d \notin P_C(D)$. Because $d \notin P_C(D)$, there is a Pareto design, $d_p \in P_C(D)$, that eclipses d. Since $P_C(D) \subseteq P_C\left(\bigcup_{j=1}^{m} \bigotimes_{i=1}^{n}\left(P_C\left(PS_j^i\right)\right)\right)$, $d_p \in P_C\left(\bigcup_{j=1}^{m} \bigotimes_{i=1}^{n}\left(P_C\left(PS_j^i\right)\right)\right)$. Therefore, both d and d_p belong to $P_C\left(\bigcup_{j=1}^{m} \bigotimes_{i=1}^{n}\left(P_C\left(PS_j^i\right)\right)\right)$. But since $d_p \prec d$, d is not a Pareto design in the set $\bigcup_{j=1}^{m} \bigotimes_{i=1}^{n}\left(P_C\left(PS_j^i\right)\right)$, leading to a contradiction. Therefore, $P_C(D) = P_C\left(\bigcup_{j=1}^{m} \bigotimes_{i=1}^{n}\left(P_C\left(PS_j^i\right)\right)\right)$. ■

4.4 Exact system Pareto generation under bounded errors

As in other areas of engineering design, computer systems designs are increasingly evaluated using extensive computer simulation, which is time-consuming and/or expensive. Designers use a range of simulation models that tradeoff accuracy and completeness for speed and cost. Also, designers use analytic models for quickly estimating performance, but at reduced accuracies. In this section, we explore how we can effectively use low cost methods for evaluation that have inherently lower accuracy without affecting the quality of the final Pareto designs. In many situations, even the best evaluation methods have some degree of inaccuracy. In such situations, the

results described in this section enable the application of hierarchical decomposition and recomposition to derive the exact system Pareto set. However, we require that the inaccuracy of an evaluation technique be bounded.

There are three sources of inaccuracies that we discuss in this section. First, the evaluation functions for the component design spaces may generate inaccurate component-level evaluation metrics. Second, the evaluation functions for the system design space may produce inaccurate system-level evaluation metrics. Finally, the decomposition may not be truly monotonic, because the component-level evaluation metrics do not cover aspects that may contribute to system-level evaluation metrics. We consider each of these three cases in detail now.

An evaluation function for a component design space may generate metrics that are within an error bound of the true metrics. A *bounded evaluation*, $E_B^i(\ )$, of a component design space, D^i, generates evaluation metrics, $E_B^i(d^i)$, that are within an error bound, $\boldsymbol{\Delta}$, of the true evaluation metrics for d^i, $E^i(d^i)$. The error bound, $\boldsymbol{\Delta}$, is a real vector of length m^i having the same dimensionality as $E^i(d^i)$. Thus, $\left|E_B^i(d^i) - E^i(d^i)\right| \leq \boldsymbol{\Delta}$. Each of the m^i components of $\boldsymbol{\Delta}$ may have a different value, reflecting the degree of inaccuracy in a particular evaluation metric. Though we treat $\boldsymbol{\Delta}$ to be a constant to limit the notational complexity, we only require that it be known once a design is evaluated. Thus, each design and its evaluation may have a $\boldsymbol{\Delta}$, computed specifically for that design and evaluation . Further, our treatment of $\boldsymbol{\Delta}$ assumes that the region of uncertainty, the region within which the true evaluation may lie, is an m^i-dimensional rectangular box. But, we believe that the results hold regardless of the shape of this region of uncertainty, provided the following definition of weak eclipsing is suitably modified.

For this subsection, we redefine the concepts of eclipsing and Pareto sets in order to account for bounded evaluation metrics. In a design space with bounded evaluations, a design, d_k^i *weakly eclipses* another design d_l^i if $E_B^i(d_l^i) - E_B^i(d_k^i) \geq 2\boldsymbol{\Delta}$. When $E_B^i(d_l^i) - E_B^i(d_k^i) \geq 2\boldsymbol{\Delta}$, we can be sure that the true evaluation of d_k^i is not more than that of d_l^i and that d_k^i weakly eclipses d_l^i under our original definition. A *bounded Pareto design* is a design that is not weakly eclipsed by any

other design in D^i. A *complete bounded Pareto set* is the set of all bounded Pareto designs. The bounded Pareto designs are not necessarily located on the Pareto surface (or curve).

In a decomposition, $\Theta(D)$, where one or more component design spaces have bounded evaluations, we first obtain the complete bounded component Paretos for each of the predicated component design spaces. Then, we apply the composition step, the system validity step and the Pareto filtering step to obtain a system-level Pareto set. Since eclipsing is defined to be weaker under bounded eclipsing, the component-level Pareto filtering may not be as effective in filtering out component designs. However, the final system-level Pareto set obtained through this process is the true system Pareto set, provided the monotonic assumptions are satisfied.

We now consider the case where the system evaluation function generates metrics that are within an error bound of the true metrics. A *bounded evaluation*, $E_B(\ )$, of a system design space, D, generates evaluation metrics, $E_B(d)$, that are within an error bound, $\boldsymbol{\Delta}$, of the true evaluation metrics for d, $E(d)$. The error bound, $\boldsymbol{\Delta}$, is a real vector of length m having the same dimensionality as $E(d)$. Thus, $|E_B(d) - E(d)| \leq \boldsymbol{\Delta}$. In a design space with bounded evaluations, a design, d_k *weakly eclipses* another design d_l if $E_B(d_l) - E_B(d_k) \geq 2\boldsymbol{\Delta}$. A *bounded Pareto design* is a design that is not eclipsed by any other design in D. The *bounded complete Pareto set* is the set of all bounded Pareto designs. The true complete Pareto set of a system is a subset of the bounded complete Pareto set.

In order to improve the efficiency of the composition process, we use bounded system evaluation functions while evaluating compositions of component Pareto designs. Pareto filtering using the inexpensive approximate evaluation yields a bounded Pareto set. Once system designs that are clearly non-Pareto are eliminated using this bounded evaluation, we use an accurate evaluation function for the remaining system designs in the bounded Pareto set in order to obtain the true Pareto. This two-step Pareto filtering at the system-level does not affect the optimality of the Pareto set that is obtained.

Consider an example where two component design spaces contain 20 Pareto designs each. Assume that an approximate bounded evaluation takes unit time whereas an accurate system evaluation takes 10 time units. We first use the approximate evaluation on all 400 designs obtained through the composition of the two Pareto sets. This step requires 400 time units. Assume we determine that 350 designs are non-Pareto, based on this bounded evaluation. We then use the accurate evaluation on the remaining 50 designs at a cost of 500 time units to determine the true Pareto set of, say, 15 designs. The total time spent in evaluating system designs is 900 time units as opposed to 4,000 time units using the straightforward approach of evaluating all 400 designs using the accurate evaluator. This example illustrates that first obtaining the bounded complete Pareto set and then evaluating only these designs using the accurate evaluator can result in substantial reductions in evaluation effort without affecting the quality of the final system Pareto set.

Finally, we consider the case where the system evaluation function is only approximately monotonic in the component-level evaluation metrics. An *approximation bound,* $\boldsymbol{\Delta}^i$, for a component design space, D^i is some positive real-valued vector, $\boldsymbol{\Delta}^i$ of length m^i. The following are defined for design spaces that are associated with a specific approximation bound. A design d_k^i $\boldsymbol{\Delta}$*-eclipses* another design d_l^i, if $E\left(d_k^i\right) < E\left(d_l^i\right) + \boldsymbol{\Delta}^i$. A $\boldsymbol{\Delta}$*-approximate Pareto design* is a design that is not $\boldsymbol{\Delta}$-eclipsed by any other design in D^i. A $\boldsymbol{\Delta}$*-approximate Pareto set* is the set of all $\boldsymbol{\Delta}$-approximate Pareto designs. In a decomposition, $\Theta(D)$, where all component design spaces are associated with an approximation bound, $\boldsymbol{\Delta}^i$, a system design d_k *componentwise* $\boldsymbol{\Delta}$*-eclipses* d_l, if one of more of the component designs d_k^i in $\oplus(d_k) = \left(d_k^1 \ldots d_k^n\right)$ $\boldsymbol{\Delta}$-eclipses the corresponding component design d_l^i in $\oplus(d_l) = \left(d_l^1 \ldots d_l^n\right)$ and the remaining component designs d_k^i are identical to the corresponding component designs d_l^i.

Definition: In an *approximately monotonic decomposition,* $\Theta_A(D)$

1. each component design space, $D^i, 1 \le i \le n$, is associated with an approximation bound, $\boldsymbol{\Delta}^i$.

2. if a system design d_k componentwise $\boldsymbol{\Delta}$-eclipses d_l, d_k eclipses d_l.

The complete Pareto set of the designs formed by the composition of the component Δ-approximate Pareto sets is the complete Pareto set for the system. This result can be proved using the same proof strategy as in Theorem 2:.

As an example of an application of this result, consider two component designs that can be composed to form a complete system, where the cost of the complete system is the sum of the costs of the component designs and an interface cost that varies arbitrarily from 0 through 50 units. In this situation, substituting a component design with another design of equivalent cost may reduce total cost by up to 50 units. The decomposition is clearly not strong or weak monotonic under our original definition and Pareto filtering at the component-level may discard components that are part of system Pareto designs. But, if we relax our Pareto filtering criteria and not discard any component design that is within 50 cost units of the actual Pareto designs, we retain all the component designs that are part of system Pareto designs.

We apply the earlier results to this example by first defining Δ^1 and Δ^2 with cost components of 50 units. With these approximation bounds, the decomposition is approximate monotonic. For each component design space, we obtain a Δ-approximate Pareto set. The Δ-approximate Pareto designs are located on a band that is 50 units wide along the cost-axis and not necessarly on a Pareto curve or surface. Though the Δ-approximate component Pareto set can be larger than the true component Pareto sets, it is likely to be much smaller than the component design spaces. We then compose the Pareto sets of the predicated validity sets as before and apply the system-level validity and Pareto filter. The resulting set is the system Pareto set.

In this subsection, we demonstrated that component-level Pareto filtering can be suitably relaxed to account for inaccuracies in component-level evaluation or in the monotonicity of the decomposition. We do require that these inaccuracies are adequately bounded.

4.5 Fast exact system Pareto generation

The number of predicated validity sets can be large, especially when the validity function enforces many interface constraints between components. The effectiveness of quality filtering decreases as the number of predicated sets increases. When each set contains a few designs, it is

unlikely that many of them are eclipsed by other designs. Also, a particular component design may appear in multiple validity sets. Thus, the composition step generates a large number of compositions and may in fact repeatedly generate the same composition. In this section, we examine fast exact methods for generating quality sets without evaluating all compositions. The methods are exact in the sense that the final quality set is not impacted by these optimizations.

There are two central ideas associated with reducing the number of compositions evaluated. First, we maintain a partial Pareto set, obtained by applying the Pareto filter on a subset of designs in the design space. Typically, these designs are the ones that have been examined so far. Second, we consider a set of compositions and estimate a lower bound on its evaluation metrics. Recall that an evaluation function $E : D \rightarrow \Re^m$ maps a design, d to $\mathbf{e} = (e_1,...,e_m)$ in the $m-$dimensional real space, $\Re^m$. A lower bound, $E_{LB}(S)$ maps a set of system designs, S, to $\mathbf{e} = \left[\min_{d \in S}(e_1(d)), ..., \min_{d \in S}(e_m(d)) \right]$. Additionally, we require that we can obtain a lower bound on the system evaluation from a set of evaluations for its component designs, i.e. $E_{LB} = f\left(E_{LB}^1, \ldots E_{LB}^n\right)$, where E_{LB}^i is a lower bound for the *i*th component .

The first step is to quickly obtain a partial Pareto set that is close to the final Pareto set. One approach is to take the union of all predicated validity sets for each component and build a single Pareto set for each component, i.e. find $P\left[\bigcup_{j=1}^{m}\left(PS_j^i\right)\right]$ for each component, $i, 1 \le i \le n$. We take the composition of all component designs and apply the system validity and quality filters. Typically, we get a set of designs covering the full range of evaluations from high performance to low cost.

The next step is to identify sets of compositions on which we can define tight lower bounds on evaluations. We wish to identify a set of compositions with similar evaluations. We pick a group of component designs, S^i, from each of the component Paretos, QS_j^i. We discuss later some methods to identify profitable groups of component designs. For each group, we evaluate a lower bound

$E^i_{LB}\left(S^i\right) = \left[\min_{d^i \in S^i}\left(e^i_1\left(d^i\right)\right), ..., \min_{d^i \in S^i}\left(e^i_m\left(d^i\right)\right)\right]$. We use these component level lower bounds to generate a system-level lower bound, E_{LB}. If this lower bound is eclipsed by the partial Pareto set then we can safely discard the entire set of compositions from further considerstaion. Otherwise, we are forced to evaluate each of these compositions, because some of them are possible system Pareto designs.

The efficacy of this approach depends on identifying compositions that lie close to each other in the system evaluation space. One approach is to pick one each of the components that have the largest impact on system evaluations. The other components are unspecified and can range over the entire set of Pareto designs. Another approach is to partition each of the component Pareto sets to smaller neighborhoods in the component evaluation space. We consider a set of neighborhoods, one from each of the components. Since each neighborhood has similar evaluations, the system evaluation bound is likely to be tight in the sense that it is close to the evaluations of the compositions in the system evaluation space.

As an example consider a system consisting of a processor and memory system, where the processor porting must match the memory porting. We have as many Pareto sets for each component as the range of ports. But we generate a single Pareto for the processor ignoring porting; similarly for the memory. We generate a partial system Pareto by taking the composition of these two Paretos, discarding those designs that do not match on porting and Pareto filtering the remaining designs. This is relatively fast compared to generating the full Pareto set using the approach in Section 4.3, especially when the range for porting is large.

If the processor has a large cost and performance impact and the memory has relatively little impact on evaluation metrics we first choose a processor design from one of the predicated Pareto sets. We consider the entire set of designs in the memory Pareto set that matches on porting. We evaluate the processor and all matching memory designs collectively and develop a lower bound evaluation for the processor and memory groups. We use this lower bound to develop a lower bound for a system design using the monotonicity assumption. If the lower bound is eclipsed by the

partial Pareto set, we move on to the next processor design; otherwise each composition is evaluated.

If the processor and memory have similar levels of impact on evaluation metrics, the previous approach is unlikely to generate a set of compositions with evaluations that are close to each other. Accordingly, a lower bound evaluation is unlikely to be tight and is unlikely to be eclipsed by the partial Pareto set.

An alternate approach is to partition each of the Pareto sets of the predicated validity sets into neighborhoods. For instance, we may choose to partition the processor Pareto sets into neighborhoods so that no design in a neighborhood is more than 10% worse in any individual metric than the lower bound evaluation for that neighborhood of designs. We estimate a lower bound evaluation for a processor neighborhood, a memory neighborhood and the set of compositions formed from these neighborhoods. If this lower bound is eclipsed by one of the designs in the partial system Pareto set, we can skip all compositions formed by the processor and memory neighborhoods. Otherwise, we evaluate each of the compositions, because some of them may be Pareto designs.

4.6 Fast approximate methods for system quality set generation

Our overall procedure generates a system quality set that is sometimes an approximation to the actual system Pareto set. In this subsection, we discuss fast approximate methods for generating system quality sets. The quality designs in the quality set are plotted on an $m-$dimensional space. A *false negative* is a system Pareto design that is not present in the quality set. A *false positive* is a non-Pareto design that appears in the system quality set. False positives and negatives may be present because of inaccurate evaluations at the system-level. But we are more interested in the effect of component-level evaluations, and therefore, we assume that system-level evaluations are accurate.

Under this assumption, false negatives are created when we discard a component design of a composition, assuming incorrectly that it cannot be part of a system Pareto. We may discard a component design either during the validity filtering step or in the quality filtering step. As we

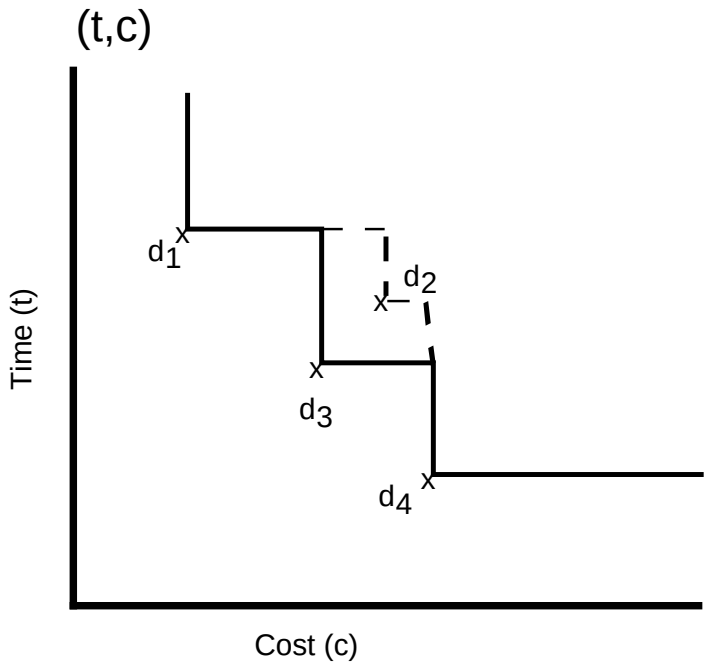

Figure 7: False positives and negatives in quality set

discuss later, our decomposition of the validity function may be approximate and inhibit compositions of component designs that indeed form Pareto designs. We may put designs that are not validity equivalent in the same predicated validity set. As a result, component design d_1 may be eclipsed by another design d_2. But d_2 may not form a valid compositions with some set of component designs from other components while d_1 does.

In Figure 7, we show a system quality set consisting of designs d_1, d_2, and d_4 whereas the actual Pareto set is $\{d_1, d_3, d_4\}$. The false negative, d_3 creates the false positive, d_2. If d_3 was present in the quality set, it would have eclipsed d_2. But since d_3 is not present, d_2 becomes a false positive. In general, we can associate each false positive with at least one such false negative. If there were no false negatives there would not be any false positives.

4.6.1 Approximations in various stages of overall procedure

There are four main steps in our overall procedure: decomposition, validity filtering, quality filtering, recomposition. Due to the nature of the system or in order to speedup the process, we

demonstrate that assumptions in each of these steps can lead to quality sets that are not Pareto sets. Finally, we discuss two steps in more detail, describing the effect of approximation assumptions on speeding up the entire process.

The input to a decomposition is a system and its parameterized design space. The decomposition step takes this input and generates simpler components. The system design space is typically specified through ranges for a set of system variables. A decomposition identifies appropriate variables for each of the components and specifies equivalent ranges for these variables. An accurate decomposition may require generating as many components as there are systems. Many of these components may only have minor variations. In order to improve the efficiency of the process, we may collapse several specialized component designs into a single generic component design, even when there are such minor variations. A final system refinement phase substitutes specialized components for these generic components. These kinds of approximations can affect the system quality set because each of the specialized component designs may have evaluations that differ from the generic component design.

The next step is validity checking which separates component designs into one of PS_j^i sets. In order to reduce the number of predicated validity sets that are generated, we may choose to use a relaxed validity checker that eventually results in invalid system compositions being generated. The system level validity checking ensures that invalid compositions are eventually removed. Also, relaxed validity checking does generate all valid systems. So, one might expect that there is no negative impact on the system quality set. But, quality filtering interacts with relaxed validity filtering to discard component designs that should be retained, as explained in the next subsection.

The third step is quality filtering at the component level. Quality filtering requires an evaluation of component designs that is ideally perfectly correlated with the system evaluation. The metrics that are available to evaluate components may not be the same as those eventually used to evaluate systems. For instance, the eventual contribution of a memory hierarchy component such as an instruction cache to overall execution time is the number of stall cycles incurred. But the stall cycles may depend on the miss penalty of the cache which depends on the second-level cache latency.

Thus we are forced to use number of misses as a component level metric which may not correlate perfectly with stall cycles. Without such correlation, we may discard component designs that are part of system Pareto designs.

In many cases, evaluation of a component requires a workload that should be generated by decomposing the system workload. The component-level workload may depend not only on the system workload but also on the designs that are chosen for other components. In this case, we have to decouple the component workload from the designs of other components. This decoupling may introduce inaccuracies in the evaluation. As a result, component designs that are actually Pareto designs may be discarded. For instance, the workload for a computer system is the application and its data set. The workload for a memory hierarchy components is an address trace. The address trace is generated by a processor design executing the application. Thus, the address trace is coupled to a processor design. We may choose a reference processor design and generate all traces using this design. But, then the evaluation of a memory hierarchy component may not generate accurate metrics when the component is actually composed with some other processor design. As a result, some memory hierarchy components that can generate system Pareto designs when composed with some non-reference processor designs may be discarded. We discuss how we maintain multiple workloads for each component to mitigate this effect in Section 5.2.

Even when accurate evaluators for the components are available they may be computationally expensive. We may choose to use approximate evaluators in an initial quality filtering of the component designs followed by an accurate evaluator in a subsequent quality filtering of the remaining designs. If there are bounds on the errors introduced by the approximate evaluators and the initial quality filtering is appropriately relaxed as discussed in Section 4.4, there is no negative impact on the quality set. However, if the errors cannot be bounded, there may be occasional false negatives and positives in the component quality set.

The final recomposition step composes component designs from predicated quality sets to form system designs, and then validity and quality filters these designs. During the composition of component designs, interconnects may have to be inserted to glue components together. These interconnects are usually not modeled as belonging to any component design even though they

contribute to final system evaluation metrics. If the contribution of the interconnects is significant and cannot be bounded, the system quality set may be negatively affected.

The number of system evaluations is proportional to the product of the sizes of the predicated quality sets. Later in Section 4.6.3, we present a method that reduces the number of system evaluations but impacts the system quality set. This method is based on the observation that the convex hull of the system Pareto set contains the most attractive Pareto designs. We develop an efficient method to construct the convex Pareto set.

4.6.2 Approximate validity filtering

The validity decomposition procedure described in Section 3.1 guarantees that any composition is a valid system. Thus, the system level validity checking is unnecessary. We may often have to or choose to relax this constraint, so that some of the compositions generated from the predicated validity sets are invalid. However, we require that all valid systems are indeed generated. The validity decomposition limits the number of compositions generated, but some compositions may be invalid. Now, the system validity checking is a necessary stage and removes any compositions that are not valid. This relaxed component-level filtering is useful in two situations.

First, the validity function is not fully decomposable. We express the validity function, $V(\ )$, as the AND of a system level validity expression, $VS(\ )$, and a component-level validity expression, $VC(\ )$ where $VC(\ )$ is decomposed as described previously, i.e.

$$V(\ ) = VS(\ ) \wedge VC(\ ) \tag{7}$$

As an example, consider a processor-memory system where the validity of a composition depends on intimate timing relationships between the two systems which cannot be expressed as a Boolean function and requires an algorithmic procedure. The compositions that are generated are limited to those that satisfy $VC(\ )$. The system validity filter rejects those compositions that do not satisfy $VS(\ )$, so that the final set of system designs all satisfy V().

Second, the validity function is fully decomposable, but doing so would generate a large number of predicated validity sets. As the number of predicated validity sets increases, the component

designs are spread over a larger number of separate spaces. The quality filtering step is unlikely to be as effective, because fewer designs are likely to be eclipsed by other designs. Moving some of the validity checking to the system level may help to greatly simplify the component-level validity checking and decrease the number of predicated validity spaces. As an example, the processor instruction fetch size, the maximum number of bytes it retrieves from the instruction stream, may be required not to exceed the instruction port width of the memory system. Typically, this width is the line size of the instruction cache size. Generally, the instruction cache line size is large for other reasons and this requirement is almost always met. However, if this condition is part of the decomposed validity function, then the number of predicated validity sets is increased by a multiplicative factor equal to the range of processor fetch sizes. For instance, let the processor fetch size be one of 2, 4, 8, 16 bytes. Assuming we initially had 8 predicated spaces each for processors and memory, the number of predicated spaces is now increased to 8*4 = 32. An alternative is to check for this condition only in the final system designs that are generated.

In this case, we generate compositions that the system level validity checking eventually finds to be invalid. However, relaxed validity checking does not inhibit the composition of valid systems. Therefore, even though validity checking is relaxed, we generate the same system validity set that would be generated even if the validity checking was not relaxed. Similarly, reducing the number of predicated validity sets also does not necessarily affect the final validity set. In the absence of quality filtering at the component level, relaxed component-level filtering does not affect the final set of system designs that are generated.

However, relaxed component-level validity filtering followed by quality filtering affects the system quality set that is generated. Let d_1 and d_2 be two Pareto designs for the *i*th component from two distinct predicated validity sets that can be composed with other component designs to form system Pareto designs. Consider combining these two validity sets into a single validity set to reduce the number of predicated validity sets. If d_1 eclipses d_2, then d_2 is discarded by the component Pareto filter. It may eventually turn out that $d_1$does not form a valid system when composed with some set of designs for each of the other components. In this case, the system with d_2 instead of d_1 may be a system Pareto design that is a false negative (not present) in the quality

set we generate. The likelihood of not generating Pareto system designs because one of its component designs was removed due to quality filtering depends partly on the amount of validity filtering at the system level. In our example of validity checking based on fetch size, few system compositions are rejected by the system level validity filter and the system quality set is close to a system Pareto set.

In general, we may also choose to use validity functions that are tighter than the original validity functions, to make the validity functions decomposable or to limit the number of validity sets that are formed. These tighter validity functions may reject certain valid designs from the predicated validity sets. As a result, some valid system designs may not be composed and the generated system quality set may not contain some of the Pareto designs.

4.6.3 Approximate recomposition using convex Pareto sets

In this subsection, we assume that the evaluation vector of all components as well as the system is two-dimensional, consisting of cost and time. The Pareto set is plotted on a graph with cost along the X-axis and time along the Y-axis. This assumption greatly improves the presentation of the concepts.

Given that the evaluation is in terms of cost and time, we assume that a set of designs is ordered in increasing cost. A *convex Pareto set* is an ordered subset of the designs such that the piecewise linear interpolant connecting each design, p_i, to the next, p_{i+1}, taking cost to be the independent variable is a convex function of cost and all designs lie on or above the piecewise linear interpolant. A *strictly convex Pareto set* is one in which the piecewise linear interpolant is strictly convex.

Let $p_i \in CP, 1 \le i \le np$ be the *i*th Pareto design in a convex Pareto, CP, where the designs are in increasing order of cost. The linear interpolant, l_i is a straight line that connects p_i to p_{i+1}. The slope of l_i is $(t(p_{i+1}) - t(p_i))/(c(p_{i+1}) - c(p_i))$. At the two end points, we assume that the slope of l_0 and l_{n_p+1} are $-\infty$ and 0 respectively. The slope of l_i monotonically increases from $-\infty$ to 0.

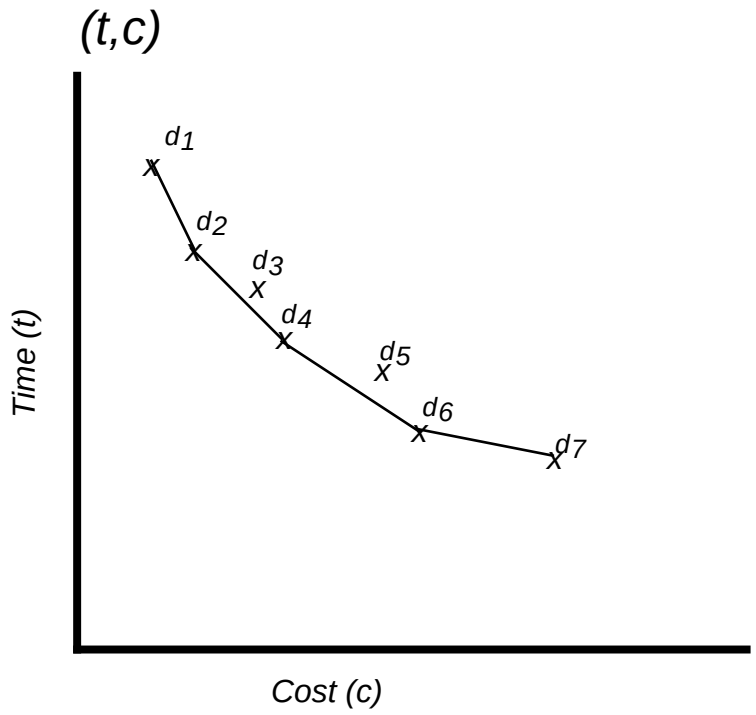

Figure 8: Convex Pareto set

Lemma 6:

The next convex Pareto point to the right of a Pareto point, p, is the composite design point $\hat{p}$, for which the line segment from p to $\hat{p}$ has the minimum (most negative) slope.

Proof: Any interpolant from p to another design with higher cost has to lie above the linear interpolant from p to $\hat{p}$. So no other design can remove $\hat{p}$ from the convex Pareto set. The designs that lie between p and $\hat{p}$ in cost are not convex Pareto designs, because they lie above the linear interpolant from p and $\hat{p}$. Therefore, $\hat{p}$ is the next convex Pareto point. ■

The convex Pareto filter, $CP(\ )$, retains a convex Pareto subset from a set of designs. The following convex Pareto filtering procedure is based on Lemma 6. Order all the designs in the set in increasing order of cost. Let p_1 be the lowest cost design and initialize i to 1. Repeatedly do the following until there are no more non-convex Pareto designs left. First, remove any designs that are eclipsed by p_i. In the time-cost plane, connect p_i to all the remaining non-convex Pareto designs,

by linear interpolants. Identify the interpolant with the lowest slope (most negative) and choose the corresponding design as the (i+1)th convex Pareto design, p_{i+1}. Increment i and repeat the process.

All the designs in Figure 8 belong to the Pareto set but the convex Pareto design set is $\{d_1, d_2, d_4, d_6, d_7\}$. Figure 8 also shows the linear interpolants connecting the convex Pareto designs. A Pareto design, p, that is not part of the convex Pareto, such as d_3 or d_5, is sandwiched between two convex Pareto designs, p_i and p_{i+1}. Factoring in the cost increment, moving from p_i to p offers a proportionately smaller improvement in time than moving from p_i to p_{i+1}. This argument can also be restated slightly differently in terms of the cost savings of moving from p_{i+1} to p versus from p_{i+1} to p. Thus, in some sense, the convex Pareto designs are the best Pareto designs.

We consider using a convex Pareto filter as the quality filter at both the component-level and system-level in the overall procedure illustrated in Figure 4. We characterize the quality of the resulting quality set in the following theorem. We prove that the system quality set is the convex Pareto set of the system design space, provided the system evaluation functions are additive in terms of the component evaluation functions. An additive evaluation function sums up the corresponding metrics at the component level. Thus the system cost is the sum of the costs of the individual components.

Theorem 3:

The convex Pareto of the system can be obtained from the convex Paretos of the individual components, provided the system time and cost evaluations are additive in terms of the time and cost of the individual components, i.e.,

$$CP(D) = CP\left(\bigcup_{i=1}^{m} \bigotimes_{i=1}^{n} \left(CP\left(PS_j^i\right)\right)\right) \tag{8}$$

Proof:

The proof is by contradiction. Assume that there is at least one convex Pareto system design, $d \in CP(D)$, that is not in the composition of convex Pareto component designs, i.e. $d \notin \bigcup_{j=1}^{m} \bigotimes_{i=1}^{n} \left(CP\left(PS_j^i\right)\right)$. By Lemma 1, $d \in \bigcup_{j=1}^{m} \bigotimes_{i=1}^{n} \left(PS_j^i\right)$ and let $d \in \bigotimes_{i=1}^{n} \left(PS_k^i\right)$ for some $1 \le k \le m$. Then, there is at least one component design, d^l, used to compose d such that $d^l \in PS_k^l$ but $d^l \notin CP\left(PS_k^l\right)$. Consider the two convex Pareto designs in $CP\left(PS_k^l\right)$ that lie immediately to the left and right of d^l in the time-cost plane. Replace d^l by each of these two convex Pareto designs and compose two system designs. Draw a linear interpolant between these system designs. Because of the additive nature of the evaluations, the original system design d is above this interpolant. These designs may not be Pareto designs themselves but the convex hull is definitely not above this interpolant. So, the original design is not a convex Pareto design, which is a contradiction. ■

The number of designs in the convex Pareto set of a component can be significantly lower than the number of designs in the Pareto set. In our example, assume that there are 10 Pareto designs for the processor and memory and 8 convex Pareto designs. The number of compositions is reduced from 100 to 64. The reduction in the number of compositions is much more than the reduction in the size of the component-level quality set. In our example, a 20% reduction in the size of the component-level quality set reduced the number of compositions by 36%. The computation cost of evaluating system designs can be the dominant cost and reducing the number of system designs that are evaluated can improve computational cost significantly. In the remainder of this subsection, we show that it is not necessary to evaluate all combinations of component-level convex Pareto designs to generate the system convex Pareto set. In our example, we show that we do not actually need to evaluate all 64 system designs but that it is sufficient to evaluate no more than 16 system designs to obtain the strict convex system Pareto set.

4.6.4 Efficient method for composing convex Pareto from convex component Paretos

In general, the required number of compositions is the product of the number of component designs in each of the *n* predicated validity sets. In this subsection, we develop a method for

composing the convex system Pareto from the convex component Paretos where the number of compositions is proportional to the sum of the number of designs in each of the *n* predicated validity sets.

The *left slope,* $s_l(k)$ of the *k*th convex Pareto design, p_k, is the slope of the $(k-1)$th linear interpolant, l_{k-1}. The *right slope,* $s_r(k)$ of the *k*th convex Pareto design, p_k, is the slope of the k th linear interpolant, l_k. Let s_l^i and s_r^i denote the left and right slopes of a convex Pareto design for the *i*th component. A set of *n* component convex Pareto designs, one each from a set of matching predicated validity sets satisfies the *slope condition* if

$$s_l^i \le s_r^j \quad \forall i, j, 1 \le i, j \le n \tag{9}$$

The slope condition holds strictly if $s_l^i < s_r^j \quad \forall i, j$.

Lemma 7:

Let the slope condition hold strictly for the set of component designs generated by the decomposition of a system Pareto design p. Consider any system design $\hat{p}$ to the right of p, i.e. $c(\hat{p}) > c(p)$, for which any of its component designs lie to the left of the corresponding component design of p, e.g. $c(\hat{p}^k) < c(p^k), 1 \le k \le n$. This system design, $\hat{p}$, is not a convex Pareto design and cannot satisfy the slope condition.

Proof: Consider $\tilde{p}$ which is the composition of designs, $\tilde{p}^i = p^i$ where $c(\hat{p}^i) < c(p^i)$and $\tilde{p}^i = \hat{p}^i$ elsewhere. The relationship between these points in the component evaluation space and in the system evaluation space is shown in Figure 9. The slopes of the lines from $\hat{p}^i$ to p^i wherever $c(\hat{p}^i) < c(p^i)$ are all more negative than the slopes of the lines from p^i to $\hat{p}^i$ elsewhere, because the slope condition holds strictly for the design p. Therefore, the slope of the line from $\hat{p}$ to $\tilde{p}$ is more negative than the slope of the line from p to $\tilde{p}$. Therefore, $\hat{p}$ lies above the interpolant between p and $\tilde{p}$. So $\hat{p}$ is not a convex Pareto point.

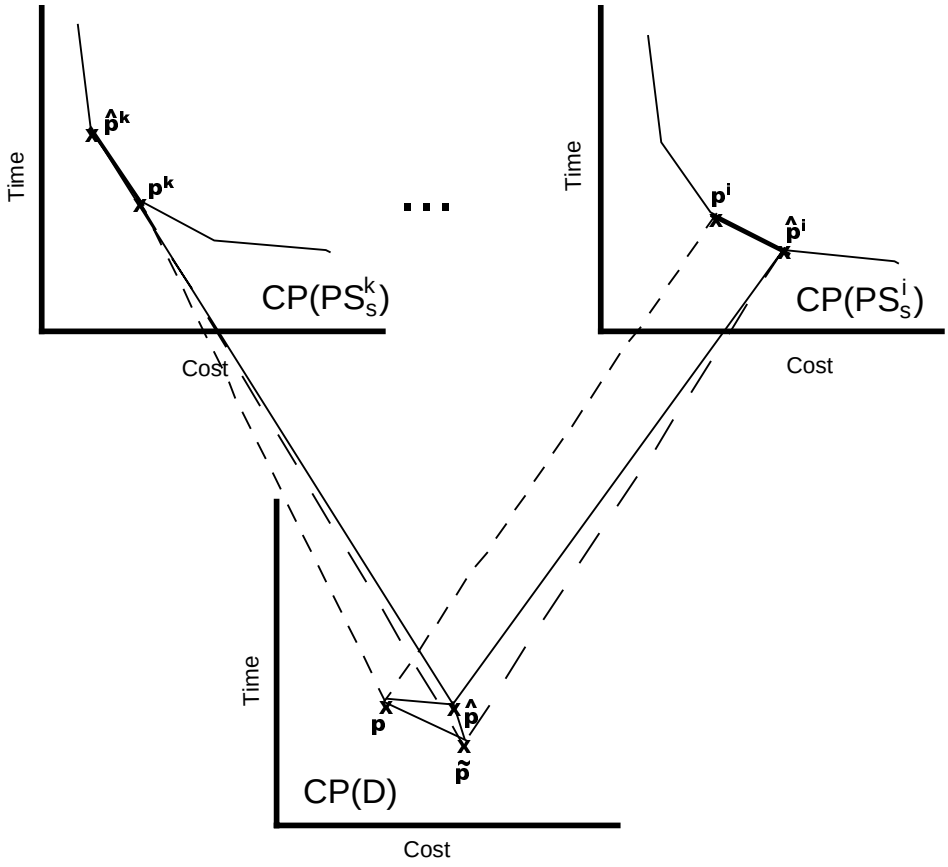

Figure 9: Convex system Pareto from convex component Paretos

With respect to the component designs that compose p, some of the component designs composing $\hat{p}$ moved right. The component evaluation space to the right of Figure 9 shows that $\hat{p}^i$ moved right from p^i. For such component designs, the left slope, $\hat{s}_l^i$, must be greater than (or identical to) the old right slope, s_r^i. Now consider the component designs composing $\hat{p}$ that moved left as shown in the component evaluation space to the left of Figure 9. The new right slope of the point that moved left, $\hat{s}_r^k$, must be more negative (or identical to) the old left slope, s_l^k. The slopes of the component designs of p are related by $s_r^i > s_l^k$ because p strictly satisfies the slope condition. Thus,

$$\hat{s}_l^i \geq s_r^i > s_l^k \geq \hat{s}_r^k \qquad (10)$$

which implies that the slope condition cannot hold for $\hat{p}$. ■

Lemma 8:

Let p be the current convex system Pareto design obtained by the composition of a set of component Pareto designs, p^i. If $s_r^k < s_r^i, \forall i \neq k$, the next convex system Pareto design (in order of increasing cost) is obtained by advancing to the next component Pareto design for component *k*. If there are several component designs with the most negative slope, then the next strictly convex system Pareto design is obtained by moving to the next convex component Pareto design for all such components.

Proof: From Lemma 7, the next convex system Pareto design cannot be composed of any component design to the left of the current convex component Pareto design. It is clear that any other composition of convex component Pareto designs, all of which are to the left (or identical to) the corresponding current convex component Pareto design will generate a system design with a less negative slope. ■

Lemma 9:

The next convex system Pareto design satisfies the slope condition.

Proof: The new left slopes of the component designs that have moved right are the same as the corresponding old right slopes. The slope relations involving the new left slopes hold because the old right slopes were the smallest. The relations involving the new right slopes remain valid because they have become larger. Therefore the slope condition continues to hold.

Theorem 4:

The composite system design $\bigotimes_{i=1}^{n}\left(p^i\right),\ p^i \in CP\left(PS_j^i\right)$ is a convex Pareto design iff the component designs, p^i, satisfy the slope condition.

Proof: The least-cost system design, obtained by the composition of the least-cost component designs from the component convex Pareto sets, is both a convex system Pareto design and satisfies the slope condition. We proceed by induction on the total cost of the system design. From Lemma 8 and Lemma 9, the next convex system Pareto design in order of increasing cost satisfies the slope condition showing necessity and the next convex system Pareto design is also the next design at which the slope condition holds showing sufficiency. ■

These results suggest an efficient algorithm for composing the convex system Pareto set from the convex component Paretos. For each set of matching convex component Paretos, we initialize pointers to the lowest cost component designs. In each step, we compose the designs indicated by the pointers to form the next convex system Pareto. Then, we identify the component designs with the smallest (most negative) right slopes and increment the corresponding pointers. We repeat until all pointers have advanced to the end of their respective convex component Paretos.

In each step, we advance at least one of the pointers to convex component Paretos. Therefore, the complexity of this algorithm is proportional to the sum of the sizes of the individual component Paretos. In contrast, the complexity of the standard composition algorithm described earlier is proportional to the product of the sizes of the individual component Paretos. In our running example, where the convex Pareto sets for the processor and memory each contain 8 designs, the number of compositions evaluated is at most 16 using this algorithm, as opposed to 64 in the standard composition algorithm.

The results of this subsection apply when there are two evaluation metrics and the system-level evaluation metric is the sum of the corresponding component-level evaluation metrics. In certain other special cases we can also show that using convex Pareto filters as quality filters does generate the convex system Pareto set. One example is where the system time is the maximum of all component times. In general, the system evaluation may not be an additive function of the corresponding component evaluations and there is no guarantee that the system quality set is the convex system pareto set. However, the system quality set is likely to be a close approximation to the convex Pareto set.

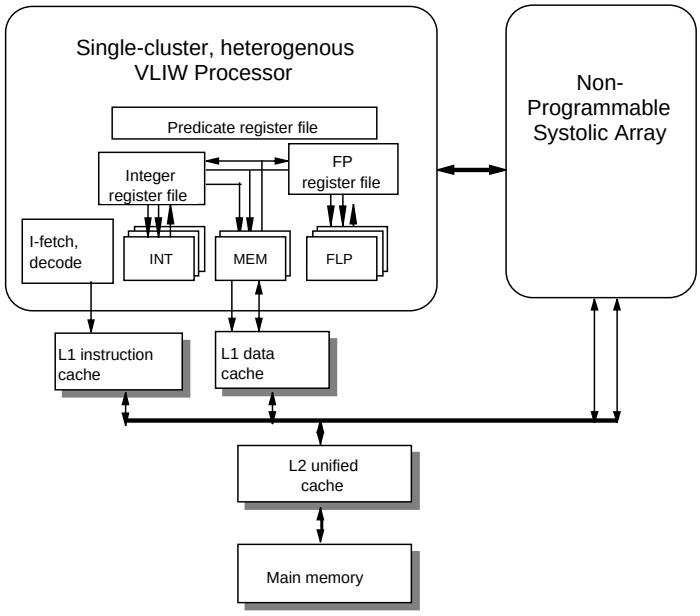

Figure 10: Overall design space

5 Application to embedded system design

In this section, we illustrate how the methodology described in the previous sections was applied to our embedded system design exploration project. First, we describe the space of embedded designs of interest and the evaluation techniques. Then, we describe our decomposition of embedded systems into components, the validity function that each composition should satisfy, and the approximations that we made in the decomposition and composition steps. Next, we describe the software framework we used to evaluate large design spaces efficiently and generate system Paretos. Finally, we present results demonstrating the systems that we generate as well as the improvements in computational time and resources stemming from our methodology.

5.1 Embedded system design space

The overall design space targeted by our system is shown in Figure 10. The design space consists of a single-cluster VLIW processor and an optional hardware accelerator in the form of a set of non-programmable systolic arrays. In Figure 11, the first three sections show the ranges for the input parameters to the design space exploration system that control the space of VLIW processors, systolic arrays and memory hierarchies respectively. Each range is specified in the following

```
// VLIW parameters
integer_units                = 1-8;1
float_units                  = 1-1;1
memory_units                 = 1-4;1
branch_units                 = 1-1;1
integer_registers            = 32-128;32
floating_registers           = 32-32;16
predicate_registers          = 32-32;16
branch_registers             = 16-16;16
predication                  = 0-1;1
hardware_speculation         = 1-1;1
machine_model                = PRUNED_IFMB
optimize_instruction_format = true

// Systolic parameters
systolic_processors          = 1-2;1
initiation_interval          =  1-8;1
systolic_cache_ports         = 1-1;1
systolic_port_width          = 32
```

```
// Memory hierarchy parameters
dc_l1_lg_cache_sets          = 7-7;1
dc_l1_cache_assoc            = 2-2;1
dc_l1_lg_cache_linesize      = 4-5;1
ic_l1_lg_cache_sets          = 7-7;1
ic_l1_cache_assoc            = 2-2;1
ic_l1_lg_cache_linesize      = 5-7;1
uc_l2_lg_cache_sets          = 8-8;1
uc_l2_cache_assoc            = 2-3;1
uc_l2_lg_cache_linesize      = 6-7;1
max_cache_trace_length       = -1.0
technology_scale             = 0.25

// VLIW design space exploration heuristics
vliwPolicy          = VLIW_LOCAL_WALK
vliwInitPolicy      = VLIW_INIT_CHEAPEST_ALL_SECTORS
vliwSelectPolicy  = VLIW_SELECT_ALL
vliwAddPolicy = VLIW_ADD_PARETO_ALL_1_NEIGHBORS
vliwSubtractPolicy = VLIW_SUBTRACT_NONE
```

Figure 11: Embedded design space exploration parameters

format: min-max:step and the set of values that a parameter assumes is $\{val : \min \le val \le \max, val = \min + i * \text{step}, \ i \ge 0\}$.

The VLIW/EPIC processor is based on the HPL-PD architecture [4] and is parameterized by the number of each type of function units, viz. the number of integer, memory and floating-point units [5, 6]. Other VLIW parameters control the size of the integer, floating-point and predicate register files, and whether the processor supports predication or speculation. Various aspects of the VLIW architecture such as the instruction format are optimized for the specified benchmark [7, 8]. The hardware accelerator (HA) consists of a set of systolic arrays where the systolic arrays are parameterized by the number of processors and their initiation interval, a measure of the performance of each processor [9, 10]. The memory system of the design is composed of a Level-1 data cache, Level-1 instruction cache, and a Level-2 unified cache [11, 12]. Each of the caches comprising the memory system is also parameterized with respect to cache size, associativity, line size, and number of ports. These three sections together provide a means for concisely specifying a system design space. The last section in Figure 11 shows the parameters that control the heuristics used to explore the VLIW processor design space, which we will discuss later.

We also require evaluation methods to evaluate an arbitrary system design. We use cost and time as the only two evaluation metrics for each system. The cost of a design is the chip area required in a specific technology to implement the entire system consisting of the VLIW, accelerator and memory hierarchy. We synthesize the VLIW and accelerators down to the RTL level, where good area models are available for the basic components such as functional units and registers. We use area models for memory hierarchy components to estimate the memory hierarchy cost. The time of a design is measured in execution cycles for a specified benchmark on a specified data set. Since we have a fully synchronous system, exactly one of the three major components, viz. VLIW, hardware accelerator, memory hierarchy, in the system is busy during the execution of the benchmark. Thus, the system execution cycles is the sum of the cycles for which the VLIW, accelerator and memory hierarchy are busy. We use profiling to estimate how many times each block of code is executed. For the loop nests that are executed on the hardware accelerator, we have a simple analytic estimate of the number of cycles per execution of the loop nest. Similarly, we examine the scheduled code to estimate the number of cycles required to execute each block of code on the VLIW processor. We combine these estimates with the profiling data to estimate VLIW and hardware accelerator cycles. In order to estimate the contribution of the memory hierarchy, we use the Cheetah cache simulator [13, 14] with the address stream generated by the VLIW and hardware accelerator and estimate the number of misses from each of the three caches, viz. L1 instruction, L1 data and L2 unified caches. The miss penalty or stalls incurred per cache miss is the access latency of the next lower level in the memory hierarchy. For instance, the miss penalty for an L1 data cache miss is the access latency of the L2 unified cache. Thus, we can derive the stall cycles for the memory hierarchy from the number of misses for each memory hierarchy component and the miss penalty. Thus, we can evaluate the cost and time for any system design.

The overall objective is to present the user with a Pareto set of system designs plotted in the time-cost space. Each Pareto design is a member of the parameterized design space and the cost and time of each design is evaluated as described above. In addition, we maintain a repository containing the intermediate code generated during the compilation and synthesis of individual components. The user may also examine the cost and time contribution of each of the major components.

A simple approach is to evaluate each system design in the design space as suggested by Figure 3. But the number of system designs increases combinatorially with the number of parameters that are used to describe the system design space. In the design space specification in Figure 11, only a few parameters assume multiple values and the ranges for these parameters are fairly small. Even then, the number of VLIW design is 256 and the number of memory hierarchy designs is 24. Assuming that the hardware accelerator is capable of accelerating two distinct loop nests, the number of hardware accelerator designs is 256. The total number of system designs is approximately 256*256*24 = 1.57 M (million).

We have used the `jpeg` benchmark extensively for evaluating and demonstrating the capabilitites of the PICO automated embedded design system. On this benchmark, evaluating each design takes approximately 30 minutes of computational time, which includes the time required for compiling and synthesizing the VLIW and hardware accelerator components and the time required for simulating the instruction, data and unified cache components respectively. The total computational effort using the design space in Figure 3 for the `jpeg` benchmark is 1.57M*0.5= 785K hours or 90 years. Clearly, we need more efficient approaches for evaluating large design spaces, even at the loss of some accuracy.

5.2 Embedded system decomposition

In this section, we describe how we decompose the system into components and the resulting validity functions. We also discuss the assumptions that we make and their effect on the final system quality set.

We use a multi-level decomposition strategy where the system is decomposed into components and some of the components are further decomposed into sub-components. The system naturally decomposes into distinct VLIW, hardware accelerator and memory hierarchy components. Each component has well-defined cost and time evaluations as described earlier. Further, the system-level cost and time evaluations are additive in terms of their corresponding component level evaluations. Under the assumption of a fully synchronous design, the system time evaluation is the sum of the three component time evaluations. Assuming that the area required for interconnecting

components is minimal, the cost evaluation is also additive. Thus, a separate system-level evaluation is unnecessary, at least in a preliminary evaluation. In our system, once we have a system Pareto, we evaluate each system's cost to account for interconnect costs. If interconnect costs are bounded, we can ensure that there is no loss of accuracy as described in Section 4.4.

When the evaluation of a component requires a workload, the workload should also be decomposed. The evaluation of a component design depends on its workload. Thus, there are component evaluations for each distinct two tuple consisting of a component design and workload. Our development of the decomposition theory did not delve into the issue of workload decomposition. But, in the context of our embedded design space exploration effort, many of the significant assumptions relate to workload decomposition.

First, we examine the workload of the VLIW designs. The workload for the VLIW processor consists of the entire benchmark, except for the loop nests that are handled by the accelerator. But, the accelerator handles any subset of the loop nests that can be accelerated. Consider a benchmark where there are n loop nests that can be accelerated. A workload may be represented by an n-bit long bit pattern, where a 1 in a bit position denotes that the corresponding loop nest is accelerated. Thus, there are a total of 2^n workloads for the VLIW processor. The number of VLIW evaluations is the product of the number of VLIW designs and the number of workloads.

Second, we examine the workload decomposition for the memory hierarchy. The actual workload for a memory hierarchy design is the address trace generated by the VLIW processor that it will eventually be composed with. But, since the memory hierarchy design may eventually be composed with almost any VLIW design, the total number of evaluations is the product of the number of VLIW designs and memory hierarchy designs. With such a decomposition, there is no reduction in the computational effort required for evaluating the memory hierarchy. In order to increase the efficiency of the evaluation, we designate a single processor design as the reference processor and evaluate all memory hierarchy designs through simulation only on traces generated by this reference processor design. Since the changes in a VLIW processor design do indeed affect the address traces and thus the stall cycles generated by the memory hierarchy, the stall cycles generated using the reference processor's trace are inaccurate for other processor designs . To

overcome this inaccuracy, we associate each processor design with a dilation coefficient that represents the extent to which each instruction block in the trace is stretched out compared to the reference processor. Analytic models are used to estimate the performance of the memory hierarchy for an arbitrarty dilation coefficient from their performance on the reference processor [11, 12]. Thus, the reference processor trace is the only workload we use for evaluating the memory hierarchy through expensive simulation. We also generate evaluations for a set of representative dilation coefficients using the reference processor evaluations and our analytic model. When we eventually compose a processor design with a memory hierarchy design, we choose the memory hierarchy evaluation whose associated dilation is closest to that of the processor design. This approach is reasonably accurate and ensures that the number of expensive memory hierarchy simulations is independent of the size of the processor design space.

We now discuss the further decomposition of the hardware accelerator and the memory hierarchy. The hardware accelerator is decomposed into individual systolic arrays, each of which accelerates a designated loop nest in the benchmark. The cost and time evaluations for the hardware accelerator are additive functions of the cost and time evaluations of its constituent systolic arrays. The memory hierarchy design space naturally decomposes into the L1 data cache, the L1 instruction cache and the L2 unified cache. The contribution of a particular cache design in terms of stall cycles cannot be estimated because that depends on the miss penalty of a cache and consequently on the access latency of the next lower level in the memory hierarchy. Therefore, instead of evaluating the time contribution of a cache design, we evaluated the number of misses incurred by a cache design. This cache design metric is independent of the final composition of a memory hierarchy design, provided the inclusion property is satisfied. In our context, the inclusion property requires that every L2 cache is large enough to always contain the contents of every L1 cache.

We now discuss the validity constraints between components. The number of data cache ports, *ndcp*, must match or exceed the number of MEM function units in the processor, *nmem*. Each processor is associated with a quantized dilation, *pdil*, corresponding to the extent to which the instructions of a particular processor design are larger than those of a reference VLIW design. Each instruction and unified cache evaluation is also associated with an assumed quantized dilation, *icdil*

and *ucdil* respectively. A processor design evaluation can be combined with a cache memory evaluation only if their dilations match. The number of unified cache ports, *uports*, must match or exceed the number of systolic ports, *sports*. Finally, each accelerator is associated with a bit vector representing the loop nests that it accelerates, *loopsAccel*, and each VLIW design evaluation is associated with a bit vector representing the loop nests that were excluded from the benchmark workload, *loopsExcl*. These two bit-vectors must match, so that the composite evaluation is accurate. Thus, the system validity function is:

$$V(\ ) = \begin{bmatrix} (ndcp \geq nmem) \wedge (pdil \equiv ucdil) \wedge (pdil \equiv icdil) \wedge \\ (uports \geq sports) \wedge (loopsAccel \equiv loopsExcl) \end{bmatrix} \qquad (11)$$

The validity function consists of five coupled terms. In our design system, each of these terms is expanded by iterating through all possible values of the partial parameters. The partial validity filters generate predicated component design spaces for each possible combination of values of the partial parameters. Thus, if the data cache design space specifies data cache ports ranging from 1 through 4, four predicated data cache design spaces are generated, each containing designs with 1, 2, 3 or 4 data cache ports. The unified cache is associated with two partial parameters, *ucdil* and *uports*. A separate predicated design space is generated for each combination of values of *ucdil* and *uports*.

We describe how the results described in this report can be applied to generate the Pareto set of computer system designs from the individual Pareto sets of processors, hardware accelerator and caches. In our design system, we first generate the individual Pareto sets for the VLIW processor, systolic arrays and the data, instruction, and unified caches. The composition step is split into two stages. In the first stage, the instruction, data and unified cache Pareto sets are combined to form a Pareto set of cache hierarchy designs. During this composition, we enforce the validity constraint that the instruction cache dilation matches the unified cache dilation. From the cache misses and area costs of the three cache designs, we obtain the stall cycles and area cost of the cache hierarchy on a particular benchmark application of interest. Also, the systolic arrays are combined to form a hardware accelerator. In the second stage, the Pareto sets of the processor, hardware accelerator and cache hierarchy designs are combined to form an overall Pareto set of embedded system designs.

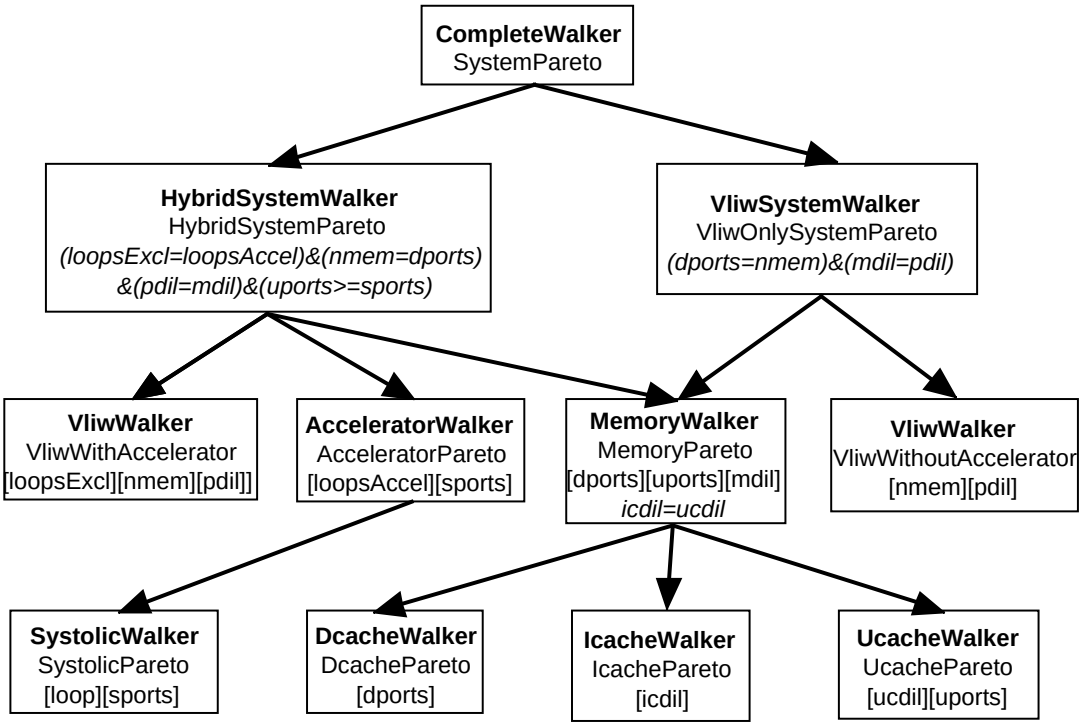

Figure 12: Walker hierarchy

5.3 Software framework

The overall design exploration system is layered and each layer contains classes, functions and/or executables that invoke other classes, functions and/or executables at a lower level layer. The software is distributed across several executables and implemented using different languages. An overview of the overall design space exploration software is provided elsewhere [11, 12]. Our focus here is on the Walkers level. The Walkers module is sandwiched between the top-most GUI (Graphical User Interface) layer, and the Pareto module. The Pareto module is responsible for implementing the Pareto filtering operation and maintaining a partial Pareto set for the Walkers level.

The Walkers module accepts an input file consisting of the design space specification as in Figure 11 and is responsible for generating Pareto sets for the overall system as well as the intermediate components. There are specialized Walkers for each component, intermediate compositions and the

final complete system. These specialized Walkers form a hierarchy paralleling the decomposition of the embedded system as shown in Figure 12. Each box shows the Walker type (in bold), the Pareto sets it generates, and the validity condition that it enforces (in italics). Each subsystem is associated with its own set of predicated validity sets. The Walkers maintain the Pareto filtered set of subsystem designs for each predicated validity set, i.e. $P\left(PS_j^i\right)$, parameterized by some set of partial parameters. The arcs between boxes show the invocation hierarchy. The system Walker, CompleteWalker invokes the subsystem Walkers, HybridSystemWalker and VliwSystemWalkers to deliver the HybridSystemPareto, a Pareto set of hybrid systems (containing the optional hardware accelerator) and the VliwOnlySystemPareto, a Pareto set of VLIW-only designs. These Walkers invoke lower-level subsystem Walkers, such as the MemoryWalker. The MemoryWalker invokes component-level Walkers such as the DcacheWalker. When invoking lower-level Walkers, the high-level Walkers provide a decomposed design space specification. For instance, the MemoryWalker provides the DcacheWalker with a specification of the L1 data cache design space.

We now discuss how the Walkers compose Paretos obtained from lower-level Walkers in the invocation hierarchy. The SystolicWalker walks over the space of systolic array designs and delivers an array of Pareto sets, parameterized by the loop nest identifier, *loop*, and the number of required memory ports, *sports.* Using these Pareto sets as input, the AcceleratorWalker generates an array of Pareto sets, parameterized by *loopsAccel*, and *sports*, where *loopsAccel* is a bit-vector representing which loops are accelerated. The MemoryWalker composes memory hierarchy designs consisting of an L1 data cache (Dcache), L1 instruction cache (Icache), and an L2 unified cache (Ucache). During the composition of such designs, it enforces the validity requirement that *icdil*, the assumed dilation for the Icache matches *ucdil*, the assumed dilation for the Ucache. The VliwSystemWalker composes system designs consisting of a VLIW processor and a memory hierarchy, enforcing the validity requirement that the processor dilation, *pdil* matches the memory hierarchy dilation, *mdil*, and that the required memory ports, *nnem*, matches the available data cache ports, *dports*. The HybridSystemWalker additionally enforces the requirement that the loops excluded during VLIW evaluation, *loopsExcl*, matches the loops handled by the accelerator, *loopsAccel*, and that the ports provided by the memory hierarchy, *uports* is not less than the ports

required by the accelerator, *sports*. The CompleteWalker applies the Pareto filter to the union of the HybridSystemPareto and the VliwOnlySystemPareto to generate the SystemPareto.

An *exhaustive* component-level Walker evaluates all the designs in its design space. In contrast, a *selective* component-level Walker uses heuristics to select and evaluate a subset of the designs in its design space. Since the evaluation of VLIW processor designs is extremely expensive and the size of the VLIW design space is also large, an exhaustive walk is often not feasible. The last section of Figure 11 contains parameters that are used to control the heuristics that are used to explore the VLIW processor design space selectively. Except for the VliwWalker, all other component-level Walkers, viz. SystolicWalker, DcacheWalker, IcacheWalker and UcacheWalker, are exhaustive.

5.4 Design exploration results

In this section, we first explore the savings in evaluation effort resulting from the decomposition strategy for the design space presented in Figure 11. Recall that an exhaustive exploration of all system designs for the `jpeg` benchmark requires an evaluation of 1.57M system designs with a computational effort of 785K hours or 90 years, as discussed in Section 5.1. The Walkers that require expensive simulation and compilation are the VliwWalker, DcacheWalker, IcacheWalker and UcacheWalker. There are 256, 2, 3, and 4 designs respectively in these four component design spaces, each requiring an evaluation effort of approximately 15 minutes. The total evaluation effort in terms of profiling, compilation and simulation time is (256+2+3+4)*0.25=66.25 hours or 2.7 days. Our decomposition approach reduces total evaluation effort by a factor of over 10,000. However, further reductions are often required. The heuristics selected using the parameters in the last section of Figure 11 reduce the total number of VLIW processor designs evaluated from 256 to 31. Now, the total evaluation effort requires around 10 hours. We further reduce the time required for a design space exploration by spreading the evaluations across multiple systems and multiple processors of a single system. These improvements are even larger for larger design spaces.

We have used our design system extensively on a range of media-intensive benchmarks and a range of design spaces. For the `ghostscript` benchmark, the systolic accelerator was not used and the system design consists of a VLIW processor and a cache memory hierarchy. On this

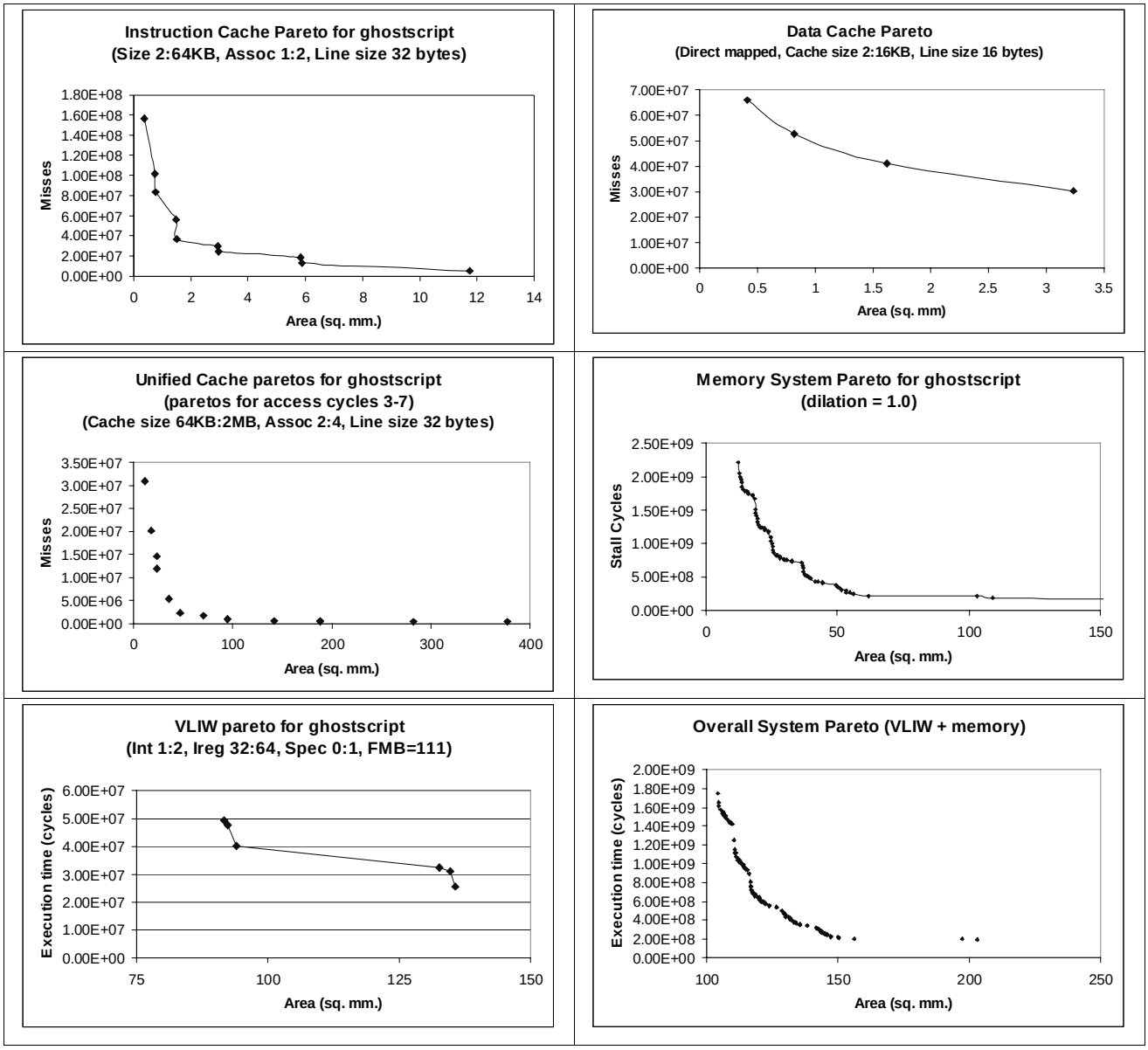

Figure 13: Pareto curves for components and overall system

benchmark, evaluation of a VLIW design requires 10 hours of profiling and compilation and evaluation of the data, instruction and unified caches requires 2, 5, 7 hours of simulation respectively. Because of the large evaluation effort required for each design, the decomposition techniques described in this report are essential to explore even relatively small design spaces.

In Figure 13, we illustrate some sample results that were obtained using our automated design system on the `ghostscript` benchmark. We present the Pareto curves for the individual caches, memory hierarchy, VLIW processor and overall system. Each point on the curve represents a Pareto design for an embedded system. Using the hierarchical decomposition approach described in this report in conjunction with related techniques reduces the simulation time required for evaluation down to 5 days.

6 Related Work

The basis for this work is that large-scale systems are often characterized by hierarchical structure and their designs usually have to meet multiple objectives that are incommensurable. As discussed in Haimes and Li [2], two well-known approaches to the analysis of systems, viz. hierarchical system theory and multiobjective optimization, have been developed to deal with these two aspects of large-scale systems. Dantzig and Wolfe [15] developed the first mathematical treatment of the decomposition of a large system into smaller, more malleable chunks in the context of linear programming problems possessing special structures. Several basic concepts of multiobjective optimization were developed in economics. Using the notion of utility, Pareto [1] introduced indifference curves as contour lines of a utility function. These curves are referred to as Pareto curves in this report and elsewhere. Zadeh [16] first presented the problem of system design with respect to several performance indices. Besides linear programming approaches, other approaches such as learning automata and evolutionary (genetic) algorithms have been developed for solving multiobjective optimization problems. Narendra and Parthasarathy [17] show that multi-level (hierarchies) of learning automatons converge to an $\varepsilon-$approximate solution of the multiobjective optimization problem. Zitzler and Thiele [18] use the designs from the partial Pareto set to assess the fitness of candidate designs in a genetic algorithm.

Haimes and Li [2] present a survey of the work in hierarchical multiobjective analysis that combines these two characteristics of large-scale systems, viz. hierarchical nature and multiobjective optimization, and classify the methods into (1) goal programming, (2) utility function, (3) weighting function, (4) trade-off method and (5) generating method approaches. The first four approaches require the *a priori* articulation of the decision-maker's preferences. The

generating methods identify the set of all non-eclipsed designs (the Pareto set); the focus of this report. Li and Haimes provide a parametric decomposition theorem for large-scale multiobjective optimization problems [19]. The original problem is decomposed into a family of subproblems by temporarily fixing the values of certain variables (parameters). Under some regularity conditions, the envelope of a family of solution sets for parameterized problems can be generated that yield the solution set of the original overall problem. We use a similar concept to decouple interactions between subsystems in the expansion of the validity function. Existing work in this area is only applicable when the design parameters are continuous, the optimization metrics are continuous well-behaved functions of the design parameters and concepts such as derivatives, dual variables and trade-off rate are well-defined. Thus, the Pareto sets are represented in a functional form. Our approach addresses the important class of design problems where the design space is discrete, optimization metrics are not continuous and may not have a functional representation, and hence concepts such as derivatives are not defined. We exploit general properties such as monotonicity to effectively decompose the system design space into smaller subsystem design spaces to greatly reduce the computational effort. We represent Pareto sets explicitly at the subsystem and system levels.

Multi-objective optimization arises in a range of problem domains. Yang and Chang [3] considered scheduling problems in semiconductor manufacturing where there are two objectives, viz. weighted tardiness and weighted cycle time. The solution techniques use a Lagrangian relaxation method to generate the approximate Pareto boundary. This problem domain does not require decomposition of the design space into subsystem design spaces. Bard [20] motivated the need for decomposing large-scale systems such as the NASA Space Station to smaller subsystems and generating a set of Pareto-optimal solutions at the subsystem level. Bard presents a sample decomposition of the Space Station design into smaller subsystems, provides a parametric integer programming algorithm for developing a Pareto set for a subsystem where the design variables are all binary. However, the decomposition techniques are not generalized beyond the Space Station example and no recomposition techniques are presented. Montusiewicz and Osyczka [21] developed a multi-stage decomposition strategy and applied it to designing machine-tool spindle systems. The first stage optimizes basic decision variables for the system, the second stage

optimizes subsystems in isolation, the third stage optimizes the overall system with respect to coordination variables between subsystems and the final stage composes solutions and applies a Pareto filter. The design parameters are continuous and relate to the physical dimensions of the spindle components. Further, the solution process maintains only a few of the candidates at each stage and there is no guarantee of optimality.

Another important area of previous research is work on automatic design of embedded systems. Hoogerbrugge and Corporaal [22] investigated the automatic synthesis of transport-triggered architectures within the MOVE framework. Their system automatically searches a template-based processor design space to identify a set of best solutions. In the SCARCE project, Mulder and Portier [23] developed a framework for the design of retargetable, application-specific VLIW processors. This framework provides the tools to tradeoff architecture organization and compiler complexity. Holmes and Gajski [24] developed the Architectural Explorer to design systems consisting of a collection of functional units and a memory hierarchy composed of explicitly managed memories of varying latency and size. Their heuristic algorithm generates a design that meets a particular performance level. They generate designs that meet a range of performance levels to generate an approximate cost/performance Pareto. Bentz *et al* [25] presented a design exploration environment that provides a uniform way to perform an exploration at the conceptual (pre-specification) stages of design. The user chooses a context from a list of choices and subsequent user requests are interpreted in that context. Guerra *et al* [26] presented a design flow where the designer refines and improves the design by applying a series of optimizations. At each step, their system provides the user with a set of optimization choices, accompanied by their likely effect on relevant optimization metrics. Peixoto *et al* [27] used a class hierarchy to represent available designs and relevant design decisions. Their system supports a systematic pruning of the large design space by the user that eventually provides the user with a small set of designs that lie in the pruned space. In contrast to previous work, our framework permits us to explore a large parameterized processor design space in conjunction with a parameterized memory hierarchy design space efficiently. Further, the results are applicable to a broader set of hierarchical discrete systems.

7 Conclusion

This report develops a formalized approach for automated hierarchical design of discrete systems. A straightforward way of generating the set of system Pareto designs is extremely expensive and time-consuming. This report introduces the concept of validity and quality filtering at the component-level to improve the efficiency of generating a set of high-quality system designs. In order to facilitate such filtering, we transform the system validity function into a suitable expanded form. Conjunctive expressions in the expanded validity function form the predicates for common and partial validity filtering at the component level. When the component-level evaluation functions are correlated to the system-level evaluation functions, quality filtering at the component level can further reduce the number of system compositions that are evaluated without affecting the quality of the system designs. We describe the application of the results in the context of an automated design of an embedded system consisting of a VLIW processor, systolic hardware accelerator and a cache memory hierarchy.

8 Acknowledgements

The authors wish to thank all the members of the Compiler and Architecture Research (CAR) group at HP Laboratories who were engaged in the PICO automated embedded system design project. In particular, we thank Greg Snider for architecting the software infrastructure of the spacewalker.

9 References

[1] V. Pareto, “Cours d'economic politique,” Rouge, Laussanne, Switzerland, 1896.

[2] Y.Y. Haimes and D. Li, “Hierarchical multiobjective analysis for large-scale systems: Review and current status,” *Automatica*, vol. 24, no. 1, pp. 53-69, 1988.

[3] J. Yang and T.-S. Chang, “Multiobjective scheduling for IC sort and test with a simulation testbed,” *IEEE Trans. Semiconductor Manufacturing*, vol. 11, no. 2, pp. 304-315, 1998.

[4] V. Kathail, M.S. Schlansker, and B.R. Rau, “HPL PlayDoh Architecture Specification: Version 1.1,” Technical Report HPL-93-80 (R.1), Hewlett-Packard Laboratories, Feb. 2000 (originally published as "HPL PlayDoh Architecture Specification: Version 1.0", Feb. 1991).

[5] M.S. Schlansker and B.R. Rau, “EPIC: Explicitly Parallel Instruction Computing,” *IEEE Computer*, pp. 37-45, Feb. 2000.

[6] M. Schlansker, et al., “Achieving high levels of instruction-level parallelism with reduced hardware complexity,” HPL-96-120, Hewlett-Packard Laboratories, Feb. 1997.

[7] S. Aditya and B.R. Rau, “Automatic architectural synthesis and compiler retargeting for VLIW and EPIC processors,” Technical Report HPL-1999-93, Hewlett-Packard Laboratories, `http://www.hpl.hp.com/techpubs/1999/HPL-1999-93.pdf`, Jan. 2000.

[8] S. Aditya, B.R. Rau, and R. Johnson, “Automatic design of VLIW/EPIC instruction formats,” Technical Report HPL-1999-94, Hewlett-Packard Laboratories, `http://www.hpl.hp.com/techpubs/1999/HPL-1999-94.pdf`, April 2000.

[9] R. Schreiber, et al., “High-level synthesis of nonprogrammable hardware accelerators,” Technical Report HPL-2000-31, Hewlett-Packard Laboratories, `http://www.hpl.hp.com/techpubs/2000/HPL-2000-31.pdf`, May 2000.

[10] R. Schreiber, et al., “High-level synthesis of nonprogrammable hardware accelerators,” Proc. ASAP, June 2000.

[11] S.G. Abraham and S.A. Mahlke, “Automatic and efficient evaluation of memory hierarchies for embedded systems,” Proc. Int. Symp. Microarchitecture, pp. 114-125, Haifa, Israel, Nov. 1999.

[12] S.G. Abraham and S.A. Mahlke, “Automatic and efficient evaluation of memory hierarchies for embedded systems,” Technical Report HPL-1999-132, Hewlett-Packard Laboratories, `http://www.hpl.hp.com/techpubs/1999/HPL-1999-132.pdf`, Nov. 1999.

[13] R.A. Sugumar and S.G. Abraham, “Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization,” Proc. ACM SIGMETRICS Conf., pp. 24-35, 1993.

[14] R.A. Sugumar and S.G. Abraham, “Multi-configuration simulation algorithms for the evaluation of computer architecture designs,” CSE-TR-173-93, CSE Division, University of Michigan, Ann Arbor, 1993.

[15] G. Dantzig and P. Wolfe, “Decomposition principle for linear programs,” *Ops. Res.*, vol. 8, pp. 101-111, 1960.

[16] L.A. Zadeh, “Optimality and non-scalar valued performance criteria,” *IEEE Trans. Automatic Control*, vol. 8, pp. 59-60, 1963.

[17] K.S. Narendra and K. Parthasarathy, “Learning automata approach to hierarchical multiobjective analysis,” *IEEE Trans. Systems, Man and Cybernetics*, vol. 21, no. 1, pp. 263-273, Jan./Feb. 1991.

[18] E. Zitzler and L. Thiele, “Multiobjective evolutionary algorithms: A comparative case study and the strength Pareto approach,” *IEEE Trans. Evolutionary Computation*, vol. 3, no. 4, pp. 257-271, Nov. 1999.

[19] D. Li and Y.Y. Haimes, “Hierarchical generating method for large-scale multiobjective systems,” *J. Optimization Theory and Applications*, vol. 54, no. 2, pp. 303-33, Aug. 1987.

[20] J.F. Bard, “A multiobjective methodology for selecting subsystem automation options,” *Management Science*, vol. 32, no. 12, pp. 1628-1641, 1986.

[21] J. Montusiewicz and A. Osyczka, “A decomposition strategy for multicriteria optimization with application to machine tool design,” *Engineering Costs and Production Economics*, vol. 20, pp. 191-202, 1990.

[22] J. Hoogerbrugge and H. Corporaal, “Automatic synthesis of transport triggered processors,” Proc. First Ann. Conf. Advanced School for Computing and Imaging, Heijen, The Netherlands, May 1995.

[23] J.M. Mulder and R.J. Portier, “Cost-effective design of application-specific VLIW processors using the SCARCE framework,” Proc. 22nd Workshop on Microprogramming and Microarchitectures, Aug. 1989.

[24] N.D. Holmes and D.D. Gajski, “Architectural exploration for datapaths with memory hierarchy,” Proc. Eur. Des. Test Conf (EDTC), pp. 340-344, 1995.

[25] O. Bentz, J.M. Rabaey, and D. Lidsky, “A dynamic design estimation and exploration environment,” Proc. 34th ACM Design Automation Conf. (DAC), pp. 190-195, June 1997.

[26] L. Guerra, M. Potkonjak, and J. Rabaey, “A methodology for guided behavioral-level optimization,” Proc. 35th ACM Design Automation Conf. (DAC), pp. 309-314, June 1998.

[27] H.P. Peixoto, et al., “The design space layer: Supporting early design space exploration for core-based design,” Proc. Design and Test in Europe (DATE), pp. 676-683, Mar. 1999.