Rapid Re-engineering of Embedded Real-Time Systems via Cost-Benefit Analysis with K-Level Diagonal Search *

Jungkeun Park, Minsoo Ryu, Seongsoo Hong † and Lucia Lo Bello‡

Abstract

This paper formulates a problem of embedded real-time system re-engineering and presents a systematic solution. Embedded real-time system re-engineering is defined as an understanding and alteration of a legacy system to guarantee newly imposed performance requirements. The performance requirements may include a real-time throughput and an input-to-output latency. The proposed approach is based on a bottleneck analysis and a nonlinear optimization. The inputs to the approach include a system design specified with a process network accompanied by task graphs and task schedules, and a new real-time throughput requirement specified as a system's period constraint. The output is a set of scaling factors that represent the ratios of performance upgrades for processing elements.

The solution approach works in two steps. First, it identifies bottleneck processes by estimating process latencies and by analyzing resource sharing among processes. It then derives a set of linear constraints from the new throughput requirement for bottleneck processes. Second, it formulates an integer nonlinear optimization problem and solves it for scaling factors with an objective of minimizing the hardware upgrade cost. Resultant scaling factors are used for cost-effective upgrades of processing elements. To efficiently find feasible solutions, we propose the k-level diagonal search algorithm which runs in a polynomial time with respect to the number of processing elements. Simulation results also confirm this assertion.

1 Introduction

There is a growing demand for techniques and tools that can assist in designing, analyzing, and debugging embedded real-time applications. In the literature, various techniques based on real-time scheduling theory and formal methods have been proposed [12, 13, 19] and many of them are implemented into software tools [2, 7]. Also, a number of commercial CASE tools have been developed and widely used [3, 17]. While most of these techniques and tools put an emphasis on the development aspect of embedded real-time systems, in practice, a great deal of effort is put into re-engineering of legacy systems. In industry, a significant number of new products are released merely as enhanced versions of old designs in their product series. For example, at SindoRico, a leading manufacturer in the Korean office automation industry, a copier code-named NT 4040 is merely an upgraded product of NT 4020. The most noticeable difference of NT 4040 from its predecessor is its improved throughput. NT 4040 is capable of copying documents at 30 cpm (copies per minute), whereas NT 4020 works only at 25 cpm.

During a product re-engineering cycle, developers are often faced with difficult tasks of cost-benefit analysis and detailed hardware/software modification. Even though existing technologies provide sophisticated design processes from an abstract behavioral level to a physical implementation level, real-time system re-engineering requires a design effort equivalent to the redesign of the entire system. Due to this difficulty, developers often rely on simple hardware upgrades, replacing all the hardware components at a time with new components. For example, the developers at SindoRico could have implemented NT 4040 by uniformly enhancing the performance of all parts, including both microcontrollers and mechanical components, in NT 4020 by 20%. However, it is fairly obvious that such a naive approach will fail in practice due to the excessive cost for re-engineering. Thus, it is inevitable for the engineers to pinpoint performance bottlenecks in the old design and carefully choose only those parts that can lead to 20% performance improvement at the least cost.

Such a task of re-engineering will get even more difficult if the original developers have been relocated to another project, or if the original systems were developed in a very ad hoc manner. Worse yet, there are very few tools to

*The work reported in this paper was supported in part by MOST under the National Research Laboratory (NRL) grant 2000-N-NL-01-C-136, by Automatic Control Research Center (ACRC), and by the Automation and Systems Research Institute (ASRI).

†School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-742, Korea. Email: sshong@redwood.snu.ac.kr.

‡Department of Computer Engineering and Telecommunications, University of Catania, Catania, Italy. Email: llobello@diit.unict.it

aid in performing such a reverse engineering, even though engineers are under tight deadline constraints for reduced time-to-market.

Re-engineering of an embedded real-time system possesses very distinct and inherent characteristics since (1) the re-engineering involves analyzing a heterogeneous distributed multiprocessor hardware platform, as an embedded real-time system often consists of multiple microcontrollers, ASIC (application specific integrated circuits) chips, and electro-mechanical components [19]; (2) software and firmware code of the underlying system has been developed and well-tested; and (3) task allocation and scheduling have been already completed. With these characteristics in mind, we propose a systematic approach to the re-engineering of embedded real-time systems.

In this paper, we present a novel methodology that allows for rapid and cost-effective re-engineering. To support rapid re-engineering, our approach chooses to upgrade only bottleneck processing elements while leaving the architecture and implementation intact. This helps significantly reduce effort for detailed re-implementation. To minimize the upgrade cost, we elaborate on determining the performance increasement for each bottleneck processing element. For rigorous cost-benefit optimization, we formulate the re-engineering problem as an integer nonlinear programming problem with a simple cost model where the cost increases as the performance does. We then present an effective heuristic algorithm that we name the k-level diagonal search algorithm. It can find feasible solutions in a polynomial time.

The overall re-engineering approach consists of two subcomponents. The first component accepts as its input a system design specified with a process network accompanied by task graphs and task schedules and an improved throughput requirement. It then identifies bottleneck processes by estimating process latencies and by analyzing resource sharing among processes. After bottleneck processes are found, it determines processing elements that should be replaced and derives a set of linear constraints from the throughput requirement. The second component accepts as its input the set of linear constraints and formulates a non-linear integer programming problem. With an objective of minimizing the hardware upgrade cost, it then solves the problem for scaling factors that represent the ratios of performance upgrades for processing elements. In doing so, it uses the k-level diagonal search algorithm. Figure 1 gives an overview of the proposed approach. As shown in the figure, the output of our approach is a set of performance scaling factors of processing elements to be replaced.

This work is an extension of our previous work in [14] where we made several assumptions to simplify the problem. In this paper we eliminate one of them. Specifically, we allow processes to share processing elements, thus complicating the bottleneck analysis. To accommodate the bottleneck analysis, we classify bottleneck processes into a *critical bottleneck* whose latency exceeds the required latency and a *correlated bottleneck* which affects the latencies of critical bottlenecks by sharing processing elements. More important contribution over [14] is that we formally redefine the optimization problem in [14] as an integer nonlinear programming problem and present a superior search algorithm, the k-level diagonal search. The old algorithm in [14] is not scalable with respect to the number of processing elements. The k-level diagonal search algorithm is not only scalable but also runs in a polynomial time. It also provides an upper bound on the number of iterations required to find an optimal solution. We analyze the complexity of the algorithm and present simulation results that confirm the analysis result.

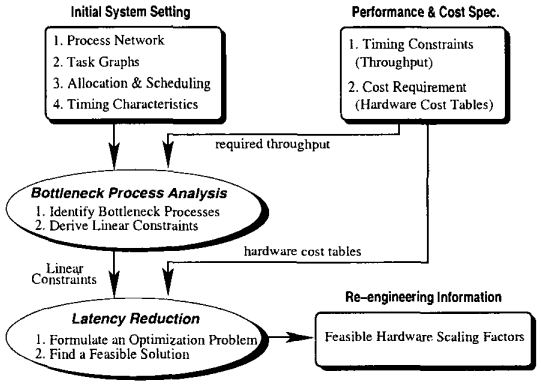


Figure 1: Overview of the approach.

1.1 Related Work

Jacobson in [4] defines the re-engineering problem as:

$$\text{Re-engineering} = \text{Reverse Engineering} + \Delta + \text{Forward Engineering}$$

The Reverse Engineering captures an understanding of the behavior and the structure of the system and Δ represents Alteration of the System. The Forward Engineering is the activity of creating new functionalities. According to this definition, our approach addresses reverse engineering and alteration of the system with an emphasis on the timing characteristics of the system. In Figure 1, the bottleneck process analysis corresponds to the reverse engineering and the latency reduction corresponds to the alteration of the system.

Existing re-engineering approaches deal with software and/or hardware components. In the literature, software re-engineering has been addressed in various directions. Scherlis et al. describe several techniques for structural re-engineering of small-scale software [15] and Karadimitriou et al. present a case study on the re-engineering of a large serial system into a distributed parallel version [6]. Jacobson

demonstrates a methodology to restructure software systems to object-oriented systems [4]. On the other hand, Stevens et al. in [16] argue that the diversity of the problem domain poses technical challenges to conventional re-engineering methodologies which are not mature yet as design methodologies. Thus, they propose to use re-engineering patterns drawn from their experience [16].

For hardware re-engineering, most research focuses on cost-effective hardware upgrades. Madisetti et al. in [11] propose a systematic technique for rapidly upgrading electronic systems. They propose to use virtual prototyping accompanied by their tools and libraries of simulatable models. Their approach is to evaluate the cost and benefit of re-engineering while performing hardware/software cosimulation. To facilitate electronic hardware re-engineering, Tummala and Madisetti in [18] show that the SoP (System on a Package) paradigm provides more architectural flexibility, thus enabling the re-engineering at a lower cost than the SoC (System on a Chip) paradigm. The re-engineering approaches in [11, 18] are similar to ours in a sense that they adopt hardware upgrades as a way of re-engineering.

This paper is organized as follows. Section 2 defines the application model of an embedded real-time system along with its timing and performance constraints. Section 3 presents as our first step the identification of bottleneck processes and derivation of linear constraints. Section 4 describes the formulation of an integer nonlinear programming problem and its solution approach. Section 5 shows the effectiveness of the heuristic algorithm with experimental results. Finally, Section 6 concludes this paper.

2 System Model and Problem Formulation

In this section, we present an application system model consisting of a process network and task graphs, along with timing characteristics associated with the application model. We then describe a digital copier as a typical example of an embedded real-time system and specify it with the presented model. Finally, we describe our problem.

2.1 Application Model

As in many other embedded system models, we use a graphical model with hierarchical abstraction [8]. Our framework renders an embedded real-time system in both a process network (PN) and a task graph (TG). Our process network is a special case of Kahn networks [5], which is a computational model where a number of concurrent processes communicate through unidirectional FIFO channels, where writes to the channel are non-blocking, and reads are blocking [8]. Specifically, the process network is a graph $G(\mathcal{P}, E)$ such that

- $\mathcal{P} = \{\sigma_1, \ldots, \sigma_u\}$ is a set of processes. A process is a mapping from one or more input streams to one or more output streams where a stream is defined as one dimensional sequence of data items.
- $E \subseteq \mathcal{P} \times \mathcal{P}$ is a set of directed edges such that $\sigma_i \rightarrow \sigma_j$ denotes precedence from σ_i to σ_j. Each process starts after it accepts inputs from all of its immediate predecessors. Once initiated, a process consumes time, thus causing input-to-output latency. Since processes are allowed to share processing elements, the input-to-output latency also includes the additional time due to the interference from other correlated processes.

Figure 2 shows an example process network. A process in a process network can be expanded into a task graph, like processes σ_3 and σ_4 in the figure. For a given task graph $G(V, E')$:

- $V = \{\tau_1, \ldots, \tau_v\}$ is a set of tasks in a process.
- $E' \subseteq V \times V$ is a set of directed edges such that $\tau_i \rightarrow \tau_j$ denotes precedence from τ_i to τ_j. Edges in a task graph have exactly the same semantics as those in a process network.

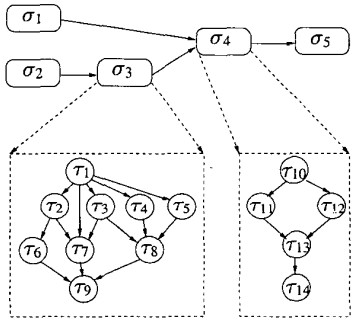


Figure 2: Process network and task graph.

The hardware aspect of an embedded system is specified with a set of processing elements $PE = \{\pi_1, \ldots, \pi_n\}$. Processing element π_i is associated with a scaling factor $\mathcal{S}_i$ which denotes the scaled performance of π_i. If there are m_i choices for processing element P_i, the scaling factor $\mathcal{S}_i$ takes m_i discrete values $\mathcal{S}_{i,1}, \mathcal{S}_{i,2}, \ldots, \mathcal{S}_{i,j_i}, \ldots, \mathcal{S}_{i,m_i}$ where $\mathcal{S}_{i,1} < \mathcal{S}_{i,2} < \ldots < \mathcal{S}_{i,j_i} \ldots < \mathcal{S}_{i,m_i}$ and $\mathcal{S}_{i,m_i} = 1.0$. The unit scaling factor $\mathcal{S}_i = 1.0$ denotes the current processing element and $\mathcal{S}_i < 1.0$ denotes a faster one. For each $\mathcal{S}_{i,j_i}$, a cost $c_i(\mathcal{S}_{i,j_i})$ of π_i is associated and c_i is a decreasing function of $\mathcal{S}_{i,j_i}$. Thus, when a certain scaling factor $\mathcal{S}_{i,j_i}$ is chosen for processing element π_i, the total hardware cost of the system is the sum of $c_i(\mathcal{S}_{i,j_i})$: $\sum_{\pi_i \in PE} c_i(\mathcal{S}_{i,j_i})$.

Task allocation Π is a mapping of tasks onto set PE such that $\Pi : V \mapsto PE$. For task τ_i, e_i denotes the worst case execution time (WCET) measured on its allocated processing element $\Pi(\tau_i)$. The WCET of a task can be measured using techniques found in the literature [1, 9].

Just as a task possesses worst case execution time, a process has a maximum input-to-output latency which is determined by the worst case execution times of tasks which belong to the process. Also, it has a period constraint which is common among all processes in the system, since all the processes should operate at the same rate. On the other hand, tasks in a process may have different rate constraints.

2.2 A Digital Copier Example

To show the expressive power of our application model, we specify a digital copier in the PN format. A digital copier is a typical example of an embedded real-time system since it possesses various electro-mechanical components and hard real-time constraints. Its major components include a scanner, a laser-beam printer, an organic photoconductive (OPC) drum, paper feeders, and transfer belts. The entire copying process can be broken down into six subprocesses, as below.

- **Feed-in**: A blank sheet is fed in from a tray via several transfer belts.
- **Exposing**: A lamp exposes the original. A reflected image is converted into a digital image, which is then stored in memory.
- **Imaging**: The stored image is retrieved and the OPC drum is charged.
- **Developing**: Toner particles are attached to the latent image on the drum and transferred to the copy paper.
- **Feed-out**: The copy is driven out from the copier.

Although the OPC drum is shared by two processes, i.e., imaging and developing, they can be parallelized in the process network architecture. Note that the rotational movement of OPC drum allows them to run simultaneously on different parts of its cylindrical surface. Figure 3 shows the PN of all copy processes and Table 1 shows their timing characteristics.

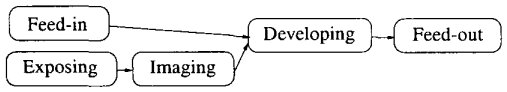


Figure 3: Digital copier example

The throughput of the digital copier is defined as the number of copies per second and is determined by the common period of processing elements. From Table 1, we see that the imaging process is a bottleneck in the copying process since it takes the longest time, 2s: it involves compute-intensive digital image processing. In this example, the period is thus 2s and throughput is 30 cpm. If we want to improve the throughput up to 40 cpm, we must reduce the latency of the imaging process down to 1.5s. Section 3 discusses

process	latency	period
Feed-in (σ_1)	0.3	2.0
Exposing (σ_2)	0.5	2.0
Imaging (σ_3)	2.0	2.0
Developing (σ_4)	1.5	2.0
Feed-out (σ_5)	0.3	2.0

Table 1: Timing characteristics of the digital copier (time in s).

identification of bottleneck processes in depth and Section 4 presents latency reduction of the bottleneck processes.

2.3 Problem Description

The objective of our approach is to identify bottleneck processes in the system and to eliminate such bottlenecks in a cost-effective manner while avoiding time-consuming hand-tuning and software redesign. To leave software intact, our approach attempts to replace bottleneck processing elements.

Inputs to this approach are:

(1) A PN and task graphs representing the underlying system.

(2) Task allocation and scheduling.

(3) A desired throughput (period) requirement.

(4) Cost tables of processing elements at various performance profiles.

Our approach works in two subsequent steps.

Step 1 It determines bottleneck processes in a PN and derives a set of linear constraints with the new period constraint.

Step 2 It formulates a non-linear optimization problem with linear constraints of step 1 and then solves the problem using a heuristic algorithm to determine processing elements for replacement.

The first step involves deriving a system of constraints according to precedence constraints imposed by the task graph structure and task scheduling. The second step involves solving these linear constraints. The objective function is to minimize the total hardware cost of the resultant system given by $\sum_{\pi_i \in PE} c_i(\mathcal{S}_{i,j_i})$. In what follows, we give a detailed description of the re-engineering approach.

3 Bottleneck Process Identification and Analysis

We present the identification of bottleneck processes and derivation of linear constraints. Bottleneck processes are classified into a *critical bottleneck* whose latency exceeds

the newly required latency and a *correlated bottleneck* whose latency does not exceed the required latency but affects the latencies of critical bottlenecks. To determine bottleneck processes, we describe the estimation of process latency and resource sharing among processes. We then derive a set of linear constraints which must be solved to reduce the latencies of bottleneck processes.

3.1 Identifying Bottleneck Processes

For a given embedded system, let $\mathcal{L}$ be the period (latency) constraint which is determined by its new throughput constraint. Then process σ_i is defined to be a *critical bottleneck* if its input-to-output latency $\mathcal{L}_i$ is greater than the period (or latency) constraint $\mathcal{L}$. If we denote the set of critical bottleneck processes by $\mathcal{P}^{\mathcal{L}}_{crit}$, it is defined by

$$\mathcal{P}^{\mathcal{L}}_{crit} = \{\sigma_i | \mathcal{L}_i > \mathcal{L}\,,\ \sigma_i \in \mathcal{P}\}.$$

Thus, to find out critical bottlenecks in the system, it is necessary to accurately estimate the worst case latencies of processes in an embedded system. As proved in [19], this is a difficult task if we allow general run-time system models; for example, it is an NP-hard problem to estimate the latency of a system possessing a set of periodic tasks which are scheduled by a preemptive priority scheduler. Thus, we make the following simplifying assumptions in our approach.

(1) Both task allocation and scheduling are off-line and static; once specified in a design, they are not changed at run time.

(2) Task scheduling is non-preemptive.

(3) All tasks in a process have the same period. When some tasks have different periods, they are unrolled for the LCM (least common multiple) of their periods and the task graph is rebuilt to incorporate the tasks newly generated through unrolling.

The above assumptions do not seriously restrict the utility of our approach since both the architecture and behavior of the embedded system are fixed during a re-engineering cycle anyway. Fixed task allocation and non-preemptive scheduling render tasks on a processing element totally ordered. This enables us to easily compute an input-to-output latency of a process by summing the worst case execution times of tasks in the sequence considering precedence constraints given in the task graph.

To reduce the latencies of critical bottleneck processes we also need to consider *correlated bottleneck* processes. The correlated bottleneck is defined as a process that either directly or indirectly affects the latencies of critical bottlenecks by sharing processing elements. A directly correlated process can affect a critical bottleneck by sharing one or more processing elements with a critical one. On the other hand, though a process shares no processing element with the critical bottleneck, it can also affect the bottleneck if it shares processing elements with another correlated bottleneck.

Let $\mathcal{P}^{\mathcal{L}}_{(corr,1)}$, $\mathcal{P}^{\mathcal{L}}_{(corr,2)}$, $\mathcal{P}^{\mathcal{L}}_{(corr,3)}$, ... be sets of correlated bottleneck processes defined by

$$\mathcal{P}^{\mathcal{L}}_{(corr,1)} = \{\sigma_i | PE_{\sigma_i} \cap PE_{\sigma_j} \neq \emptyset\,,\ \sigma_i \in (\mathcal{P} - \mathcal{P}^{\mathcal{L}}_{crit})\,,\ \sigma_j \in \mathcal{P}^{\mathcal{L}}_{crit}\}$$

$$\mathcal{P}^{\mathcal{L}}_{(corr,i)} = \{\sigma_i | PE_{\sigma_i} \cap PE_{\sigma_j} \neq \emptyset\,,\ \sigma_i \in (\mathcal{P} - \mathcal{P}^{\mathcal{L}}_{crit} - \bigcup_{j=1}^{i-1} \mathcal{P}^{\mathcal{L}}_{(corr,j)})\,,\ \sigma_j \in \mathcal{P}^{\mathcal{L}}_{(corr,i-1)}\}$$

where PE_{σ_i} denotes a set of processing elements that run the tasks of σ_i. The set of bottleneck processes $\mathcal{P}^{\mathcal{L}}$ is given as below.

$$\mathcal{P}^{\mathcal{L}} = \mathcal{P}^{\mathcal{L}}_{crit} \cup (\bigcup_{i=1}^{\infty} \mathcal{P}^{\mathcal{L}}_{(corr,i)})$$

Walk-through Example: As a walk-through example, we consider the digital copier described in Section 2.2. Suppose that new latency requirement $\mathcal{L}$ is 15 time units. From Table 1, we see that the imaging process is a critical bottleneck for the new design, since it has the maximum latency of 20 time units, which exceeds the required latency.

Table 2 shows the resource allocation of the digital copier. Since the developing process σ_4 shares processing element π_1 with the imaging process σ_3, the developing process σ_4 is a directly correlated bottleneck of the imaging process σ_3. We see from Table 2 that no other processes share the processing elements with the developing process σ_4. Thus, bottleneck processes found in our digital copier example are the imaging process σ_3 and developing process σ_4.

process	processing elements
Feed-in (σ_1)	π_4
Exposing (σ_2)	π_5
Imaging (σ_3)	π_1, π_2
Developing (σ_4)	π_1, π_3
Feed-out (σ_5)	π_4, π_6

Table 2: Resource allocation of digital copier.

3.2 Deriving Linear Constraints

Once the bottleneck processes are identified, we derive a system of constraints using precedence constraints imposed by the task graphs and task scheduling. To facilitate the constraint derivation process, we augment our task graph model by adding precedence edges imposed by task scheduling. Specifically, we add an extra edge $\tau_j \rightarrow \tau_i$ if no such edge exists in the original task graph and if τ_i is scheduled next to τ_j on the same processing element.

Let $s(\tau_i)$ and $f(\tau_i)$ denote the start and finish time of τ_i, respectively. For each critical bottleneck process, a constraint imposed on its tail task τ_i is derived, as below.

$$f(\tau_i) \leq \mathcal{L} \iff \max_{\tau_j \in Pred(\tau_i)} \{f(\tau_j)\} + e_i \mathcal{S}(\Pi(\tau_i)) \leq \mathcal{L} \qquad (1)$$

where $Pred(\tau_i)$ is a set of τ_i's immediate predecessors in the augmented task graph. Eq. (1) means that τ_i can start after all of its predecessors finish and that finish time τ_i is determined by the scaling factor of processing element $P(\tau_i)$. If $\mathcal{S}(\Pi(\tau_i)) < 1.0$, τ_i will finish earlier than it does on the currently hardware platform.

By recursively substituting $f(\tau_j)$ with $s(\tau_j) + e_j\mathcal{S}(\Pi(\tau_j))$ for all $\tau_j \in Pred(\tau_i)$ in Eq. (1) in a reverse topological order, we can eliminate start and finish time functions from Eq. (1). Since e_i is constant, we end up with a system of linear constraints possessing only $\mathcal{S}(\Pi(\tau))$ as variables.

Walk-through Example: Recall the digital copier example. Figure 4 (A) shows the task schedules on three processing elements, P_1, P_2, and P_3. Figure 4 (B) shows the augmented task graph of the imaging process σ_3 and that of the developing process σ_4, where dashed arrows denote added precedence constraints captured from the task schedules. In Figure 4 (B), the shaded area shows the task set allocated to P_1 which is shared by the both processes. Since the task graphs are hosted by three processing elements P_1, P_2 and P_3 as in Figure 4 (A), we have three scaling factors $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$, respectively.

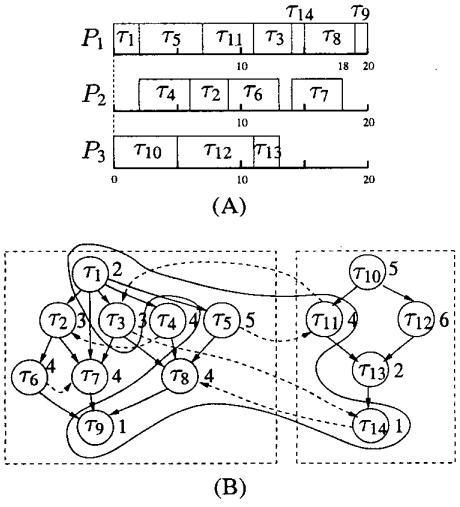


Figure 4: (A) Schedule lists and (B) augmented task graph.

As an illustration, we derive a constraint for τ_9 which is the tail task of the critical bottleneck σ_3, as below.

$$\max\{f(\tau_6), f(\tau_7), f(\tau_8)\} + e_9\mathcal{S}_1 \leq 15$$

Note that $\mathcal{S}_1 = \mathcal{S}(\Pi(\tau_9))$. Since there exists an augmented edge from τ_6 to τ_7 and this implies $f(\tau_6) < f(\tau_7)$, the above constraint reduces to $\max\{f(\tau_7), f(\tau_8)\} + e_9\mathcal{S}_1 \leq 15$. We split this constraint to eliminate $\max\{\cdot\}$, as below.

$$f(\tau_7) + e_9\mathcal{S}_1 \leq 15 \text{ and } f(\tau_8) + e_9\mathcal{S}_1 \leq 15$$

By recursively computing finish times of tasks in a reverse topological order, we have the following inequalities.

$$\begin{aligned}
(e_1 + e_9)\mathcal{S}_1 + (e_4 + e_2 + e_6 + e_7)\mathcal{S}_2 &\leq 15 \\
(e_1 + e_5 + e_{11} + e_3 + e_9)\mathcal{S}_1 + e_7 \cdot \mathcal{S}_2 &\leq 15 \\
(e_1 + e_5 + e_{11} + e_3 + e_{14} + e_8 + e_9)\mathcal{S}_1 &\leq 15 \\
(e_1 + e_8 + e_9)\mathcal{S}_1 + e_4 \cdot \mathcal{S}_2 &\leq 15 \\
(e_1 + e_5 + e_{11} + e_{14} + e_8 + e_9)\mathcal{S}_1 + e_{13} \cdot \mathcal{S}_3 &\leq 15 \\
(e_{11} + e_3 + e_9)\mathcal{S}_1 + e_7 \cdot \mathcal{S}_2 + e_{10} \cdot \mathcal{S}_3 &\leq 15 \\
(e_{11} + e_3 + e_{14} + e_8 + e_9)\mathcal{S}_1 + e_{10} \cdot \mathcal{S}_3 &\leq 15 \\
(e_{11} + e_{14} + e_8 + e_9)\mathcal{S}_1 + (e_{10} + e_{13})\mathcal{S}_3 &\leq 15 \\
(e_{14} + e_8 + e_9)\mathcal{S}_1 + (e_{10} + e_{12} + e_{13})\mathcal{S}_3 &\leq 15
\end{aligned}$$

By substituting e_is in the above constraints with values in Figure 4 (A) and by eliminating redundant inequalities, we have eight inequalities as below.

$$3\mathcal{S}_1 + 15\mathcal{S}_2 \leq 15,\ 15\mathcal{S}_1 + 4\mathcal{S}_2 \leq 15,\ 20\mathcal{S}_1 \leq 15,$$
$$17\mathcal{S}_1 + 2\mathcal{S}_3 \leq 15, 8\mathcal{S}_1 + 4\mathcal{S}_2 + 5\mathcal{S}_3 \leq 15,\ 13\mathcal{S}_1 + 5\mathcal{S}_3 \leq 15,$$
$$9\mathcal{S}_1 + 7\mathcal{S}_3 \leq 15,\ 5\mathcal{S}_1 + 13\mathcal{S}_3 \leq 15.$$

4 Latency Reduction of Bottleneck Processes

For a set of derived linear constraints, we find feasible scaling factors that reduce the latencies of bottleneck processes in a cost-effective way. This optimization problem makes a nonlinear integer programming problem and is solved by a heuristic search algorithm we propose in this section. The result is a set of scaling factors which are used for the upgrade of processing elements.

4.1 Formulating Nonlinear Optimization Problem

After linear constraints are derived, we find a feasible solution Γ – an n-dimensional column vector of scaling factors $[\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n]^T$ – that satisfies the linear constraints while minimizing the total hardware cost given by $C(\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n) = \sum_{P_i \in PE} c_i(\mathcal{S}_i)$. Since every scaling factor $\mathcal{S}_i$ takes discrete values, each value can be represented by relabeling $\mathcal{S}_i$ with j_i in an increasing order. Let Λ be an n-dimensional column vector $[j_1, j_2, \ldots, j_n]^T$ which is associated with $[\mathcal{S}_{1,j_1}, \mathcal{S}_{2,j_2}, \ldots, \mathcal{S}_{n,j_n}]^T$ by $\Gamma = S(\Lambda)$. Suppose that we have n scaling factors and m linear constraints which are derived from Eq. (1). The problem now can be transformed into the following integer nonlinear programming form:

$$\begin{aligned}
&\text{minimize } C(S(\Lambda)) \\
&\text{subject to (linear constraint) } AS(\Lambda) \leq B, \text{ and} \\
&\text{subject to (range constraint) } L \leq \Lambda \leq U,
\end{aligned}$$

where $C(S(\Lambda))$ decreases with respect to every j_i, A is an $m \times n$ matrix whose entries are the coefficients of scaling

factors in the derived linear constraints, B is an $n \times 1$ matrix whose entries are the same as the required latency $\mathcal{L}$, L is an n-dimensional column vector whose entries are the lower bounds of j_i, and U is an n-dimensional column vector whose entries are the upper bounds of j_i. Thus, $AS(\Lambda) \leq B$ is a matrix expression for the set of linear constraints and $L \leq \Lambda \leq U$ is a matrix expression for the range constraint of j_i.

The problem formulated above can be relaxed by dropping the integrality restriction and the resulting problem can be solved by using nonlinear programming techniques such as Primal methods, Penalty and barrier methods, and Lagrange methods [10]. However, this approach requires rounding-off to find the nearest integer solutions, yielding suboptimal solutions. Another naive approach is to compute the cost of every feasible solution and select the one with the minimum cost. This requires examining all possible combinations of scaling factors. If each scaling factor $\mathcal{S}_i$ can take m_i values, the search space is enormous possessing $m_1 \times m_2 \times \ldots \times m_n$ elements. To expedite the search process, we present a heuristic algorithm in the next section.

Walk-through Example: Revert to the digital copier example in section 3.2. The linear constraints derived in the previous section gives the linear constraint matrices A and B.

$$A = \begin{pmatrix} 3 & 15 & 0 \\ 15 & 4 & 0 \\ 20 & 0 & 0 \\ 17 & 0 & 2 \\ 8 & 4 & 5 \\ 13 & 0 & 5 \\ 9 & 0 & 7 \\ 5 & 0 & 13 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 15 \\ 15 \\ 15 \\ 15 \\ 15 \\ 15 \\ 15 \\ 15 \end{pmatrix}$$

The cost function $C(\cdot)$ is determined by the cost tables given in Table 3. For example, $\Lambda = [2, 2, 3]^T$ maps to $C(S(\Lambda)) = C([0.5, 0.7, 0.6]^T) = 50 + 70 + 150 = 270$. The cost tables also give the range constraint matrices L and U.

$$L = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \quad \text{and} \quad U = \begin{pmatrix} 4 \\ 4 \\ 6 \end{pmatrix}$$

4.2 Search Heuristics

We present two heuristics to find feasible solutions to the optimization problem while effectively reducing the search time. The first heuristic exploits the monotonicity of cost function, i.e., the cost decreases as the scaling factor increases. This leads to a geometric property that the local optimum is found at the end of diagonal line traversing an n-dimensional hypercube space. As seen from Figure 5, the search starts from the origin and incrementally examines each point along the diagonal until any of the linear constraints are violated. This diagonal search can greatly reduce the search time since it explores a one-dimensional line without visiting all possible points in the n-dimensional space.

$\mathcal{S}_{1,i_1}$	$\mathcal{S}_{1,1}$	$\mathcal{S}_{1,2}$	$\mathcal{S}_{1,3}$	$\mathcal{S}_{1,4}$
scaling factor	0.3	0.5	0.8	1.0
cost	100	50	20	0

Cost table of P_1

$\mathcal{S}_{2,i_2}$	$\mathcal{S}_{2,1}$	$\mathcal{S}_{2,2}$	$\mathcal{S}_{2,3}$	$\mathcal{S}_{2,4}$
scaling factor	0.5	0.7	0.9	1.0
cost	150	70	30	0

Cost table of P_2

$\mathcal{S}_{3,i_3}$	$\mathcal{S}_{3,1}$	$\mathcal{S}_{3,2}$	$\mathcal{S}_{3,3}$	$\mathcal{S}_{3,4}$	$\mathcal{S}_{3,5}$	$\mathcal{S}_{3,6}$
scaling factor	0.3	0.5	0.6	0.7	0.9	1.0
cost	300	200	150	100	50	0

Cost table of P_3

Table 3: Cost tables for P_1, P_2, and P_3

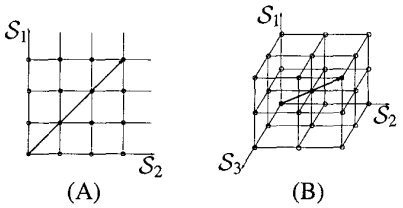


Figure 5: Diagonal search (A) in 2-dimensions and (B) in 3-dimensions.

The diagonal search can, however, examine only a hypercubic space while the problem space is arbitrary but convex[1]. To cover the entire space, the remaining space is split into disjoint subspaces and the diagonal search is iteratively applied to each subspace. Our second heuristic uses the tangent planes of the hypercube to partition the remaining space, i.e., the outer space of the hypercube is divided by its tangent planes. As seen from Figure 6, the remaining space is cut in turn by each tangent plane parallel to $\mathcal{S}_i = 0$, yielding n subspaces. Since each cutting plane is parallel to $\mathcal{S}_i = 0$, generated subproblems are of the identical form with the original problem.

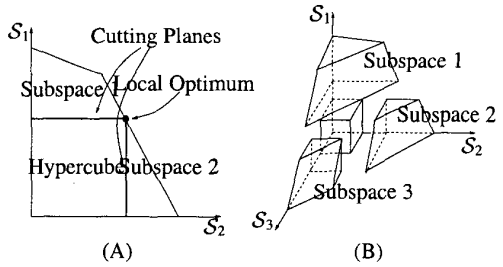


Figure 6: Partition by tangent planes: (A) in 2-dimensions, and (B) in 3-dimensions.

[1] Positive coefficients in the linear constraints lead to a convex polytope.

4.3 K-level Diagonal Search Algorithm

Our divide-and-conquer strategy generates a tree of subproblems and thus has several choices for the tree traversal policy. Let $H_{p,q}$ be the q^{th} hypercube space examined at p^{th} level, $G_{p,q}$ be the q^{th} subproblem space generated at p^{th} level, and $L_{p,q} \leq \Lambda_{p,q} \leq U_{p,q}$ be the range constraint for $G_{p,q}$. Note that every subproblem is subject to the same linear constraint with the original problem and is different only in the range constraint. Following this notation, the original problem space is represented by $G_{1,1}$. Figure 7 shows a problem tree in 2-dimensions.

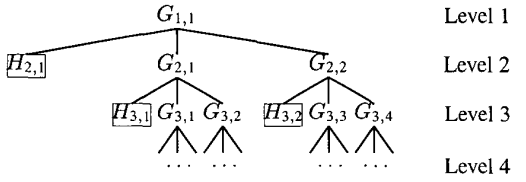


Figure 7: Problem tree in 2-dimensions.

Since the problem tree can grow arbitrarily, we want to avoid exploring indefinitely deep paths. This gives rise to the breadth-first search policy. Based on the breadth-first search, the algorithm finds a suboptimal solution exploring subproblems in levels and improves the solution going into deeper level. Limiting the search level allows us to control the search time while compromising the optimality. We call the search algorithm *k-level diagonal search* which limits the search level below k. Now we formally describe our basic heuristics as subprocedures which are called by the k-level diagonal search algorithm. The first procedure describes the diagonal search heuristic and the second procedure describes the partition heuristic.

(P1) Find a maximum integer $\delta_{p,q}$ such that $AS(L_{p,q} + \delta_{p,q}E) \leq B$ and $L_{p,q} + \delta_{p,q}E \leq U_{p,q}$, where E denotes the sum of unit column vectors E_i, i.e., $E = E_1 + E_2 + \ldots + E_n = [1, 1, \ldots, 1]^T$.

(P2) Find n disjoint subproblems ($1 \leq j \leq n$).

$$(j) \quad L_{p+1,n(q-1)+j} \leq \Lambda_{p+1,n(q-1)+j} \leq U_{p+1,n(q-1)+j}$$
$$L_{p+1,n(q-1)+j} = L_{p,q} + (\delta_{p,q} + 1)E_j \text{ and}$$
$$U_{p+1,n(q-1)+j} = U_{p,q} - \sum_{i=1}^{j-1}(U_{p,q}^T - L_{p,q}^T)E_iE_i$$
$$+\delta_{p,q}\sum_{i=1}^{j-1}E_i$$

The procedure P1 executes the diagonal search by finding the maximum increment of the scaling factor index. The index vector Λ is increased uniformly in the direction $E = [1, 1, \ldots, 1]^T$ and it is checked for the linear constraint $AS(\Lambda) \leq B$. The procedure P2 provides the partition rule required for subproblem generation. Each subspace is represented by range constraint vectors L and U.

There are two conditions for stopping subtree generation at a problem node. The first condition is $\delta_{p,q} \not\geq 0$. This implies that there exist no feasible solutions in $G_{p,q}$ and the algorithm does not generate any subproblems of $G_{p,q}$. The second condition is $L_{p,q} \not\leq U_{p,q}$, which indicates that $G_{p,q}$ is empty. The algorithm does not generate subproblems in this case, either. As a result, the entire search process terminates when the search level reaches k or when any further subproblems are not generated.

Walk-through Example: We revert to our walk-through example. Table 3 gives cost tables for P_1, P_2, and P_3. Let p be the level of search, then the proposed algorithm iteratively processes each subproblem, as follows.

$< p = 1 >$ $G_{1,1}$: $L_{1,1} = [1, 1, 1]^T$ and $U_{1,1} = [4, 4, 6]^T$, $\delta_{1,1} = 1$ and $C(\Lambda) = 380$

$< p = 2 >$ $G_{2,1}$: $L_{2,1} = [3, 1, 1]^T$ and $U_{2,1} = [4, 4, 6]^T$, $\delta_{2,1} \not\geq 0 \rightarrow$ no feasible solution

$G_{2,2}$: $L_{2,2} = [1, 3, 1]^T$ and $U_{2,2} = [2, 4, 6]^T$, $\delta_{2,2} = 0$ and $C(\Lambda) = 430$

...

$< p = 7 >$ ...

$G_{7,324}$: $L_{7,324} = [1, 1, 1]^T$ and $U_{7,324} = [4, 4, 6]^T$, $\delta_{7,324} = 1$ and $C(\Lambda) = 130$

...

$G_{7,486}$: $L_{7,486} = [1, 1, 1]^T$ and $U_{7,486} = [4, 4, 6]^T$, $\delta_{7,486} = 1$ and $C(\Lambda) = 130$

Note that $G_{2,1}$ has no feasible solutions and thus does not spawn any subproblems – $G_{3,1}$, $G_{3,2}$, and $G_{3,3}$ are not generated. The optimal solutions are found in $G_{7,324}$ and $G_{7,486}$ at level 7.

4.4 Analysis of K-level Diagonal Search Algorithm

The running time of the k-level diagonal search algorithm mainly depends on the number of checks for the linear constraints $AS(\Lambda) \leq B$ (multiplication and comparison of matrices). From this viewpoint, the approximate running time can be represented by $\sum_{p=1}^{k}\sum_{q=1}^{n^p}(\delta_{p,q} + 1)$ where $(\delta_{p,q} + 1)$ is the number of checks for linear constraints. The worst case occurs when every diagonal search fails to explore a hypercube space and examines only one point, i.e., $\delta_{p,q} = 0$. This case generates the largest problem tree and the algorithm exhaustively examines candidates one by one. By substituting $\delta_{p,q} = 0$, the running time for the worst case

is

$$\sum_{p=1}^{k}\sum_{q=1}^{n^p}(\delta_{p,q}+1) = \sum_{p=1}^{k}\sum_{q=1}^{n^p}1 = \sum_{p=1}^{k}n^p = \frac{n^{k+1}-n}{n-1} = O(n^k) \quad (2)$$

where $k \geq 1$.

In the analysis above, the search level k is bounded from above if the problem space is finite. To compute the upper bound of k, we define a metric M which can capture the size of the problem space. Let $M_{p,q}$ be a metric – distance or diameter – for the problem space $G_{p,q}$, which is defined as below.

$$M_{p,q} = ||U_{p,q} - L_{p,q}|| = (U_{p,q}^T - L_{p,q}^T)E \quad (3)$$

The k-level diagonal search always decreases the metric value by at least one as the iteration proceeds. The algorithm eventually terminates when the metric M becomes zero, which implies an empty space, i.e., $L_{p,q} = U_{p,q}$. Thus, the upper bound of k for the original problem $G_{1,1}$ is $(M_{1,1}+1)$ since a problem with zero metric requires at least one level of search. This upper bound is referred to a *guarantee level* in that it guarantees finding the optimal solution if the search level is increased up to $(M_{1,1}+1)$.

Walk-through Example: We compute the guarantee level for the digital copier example. The matrices of the range constraint for $G_{1,1}$ are $L_{1,1} = [1,1,1]$ and $U_{1,1} = [4,4,6]$. By using Eq. (3), its metric is $M_{1,1} = (4-1)+(4-1)+(6-1) = 11$ and thus the guarantee level is $11+1 = 12$. This value guarantees that the search level does not exceed 12 for searching the entire space.

5 Performance Evaluation

We evaluate the effectiveness of the proposed search algorithm by a series of simulations with randomly generated workload. For performance comparison, we have implemented the k-level diagonal search (KD) and a brute-force search (BF). The latter exhaustively examines all candidate solutions. Test problem sets were synthesized based on a random number generator for distinct dimension variables (the number of PEs) ranging from 3 to 9. It is necessary to limit the number of PEs below 10 since the BF algorithm needs to perform exhaustive search. The entries of linear constraint matrices (A and B) were chosen in the range $[1, 1000000]$. Cost tables were obtained by sorting random numbers in $[1, 100]$ to meet the monotonicity condition.

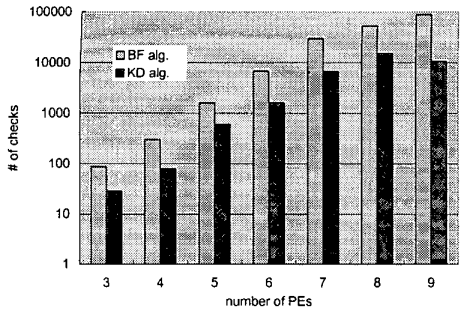
To assess average case performance, two performance metrics were used: (1) the number of checks for linear constraints and (2) the maximum search level for finding optimal solutions. The former was chosen as a measure to represent the running time of the algorithm. We counted the number of checks while running the two algorithms. We draw the result in Figure 8 (A), which shows that the KD algorithm greatly outperforms the BF algorithm. While the running time of the BF algorithm exponentially increases as the workload gets heavier, that of the KD algorithm increases much slowly and even decreases when the number of PEs exceeds eight. This anomaly can occur because the performance of the KD algorithm depends more on the shape of the feasible solution space than on the size of the problem. Due to the diagonal search strategy, the KD algorithm finds the optimal solution faster if the shape of the feasible space is closer to a hypercube.

The second performance metric is the maximum search level required for a complete search. Figure 8 (B) shows the comparison between the maximum search levels and guarantee levels. Guarantee levels were computed by using Eq. (3). On the average, the actual search level was around the half of the guarantee level. This confirms our analysis result that the search level cannot exceed the guarantee level. It is consistent with the result of the first experiment that the search level does not necessarily increase as the number of PEs increases whereas the guarantee level does. Particularly, the KD algorithm results in the minimal search level when the number of PEs was nine. These results also support our assertion that the performance of the KD algorithm depends dominantly on the shape of the feasible space.

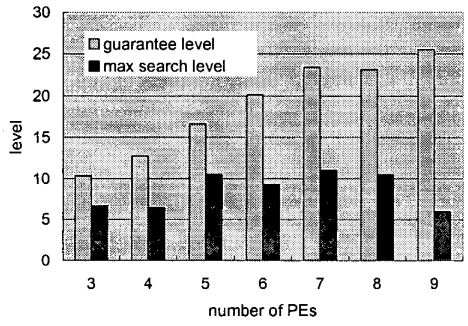
6 Conclusions

The contributions in this paper are mainly three-fold. First, we have formulated an embedded real-time system re-engineering as performance optimization of the system. While we have focused on re-engineering of hardware components in this paper, our formulation can be easily generalized to incorporate software re-engineering when appropriate cost tables are given. Second, we have proposed a systematic solution approach to the re-engineering problem. Unlike other approaches based on critical path optimization, our approach does not lead to excessive optimization since it is based on the trade-off analysis between cost and performance. It appears that this approach can be employed even at the earlier stage of system design. Third, we have presented a heuristic search algorithm which runs in a polynomial time with respect to the number of processing elements. We have also derived the guarantee level as the limit of search levels required to find an optimal solution.

There are several future research directions. First, we intend to eliminate the assumption imposed on task scheduling. In Section 3, we assumed that the task scheduling is static and non-preemptive. In fact, dynamic and preemptive scheduling incurs several complications. If we allow dynamic and preemptive scheduling, it will become extremely difficult to tightly estimate latencies of processes since tasks can arbitrarily interleave. Second, another interesting issue is the introduction of different timing requirements on a process network such as the input-to-output response time.



(A)



(B)

Figure 8: (A) Performance comparison between KD algorithm and BF algorithm and (B) guarantee level vs. maximum search level.

Unlike the throughput requirement, reducing end-to-end response times requires a trade-off analysis among process latencies. This problem is conjectured to be NP-hard and will require an efficient heuristic approach that can decompose the response time into process latencies.

References

[1] R. Arnold, F. Mueller, and D. Whalley. Bounding worst-case instruction cache performance. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 172–181, December 1994.

[2] G. Arora and D. Stewart. A tool to assist in fine-tuning and debugging embedded real-time systems. In *ACM Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 73–92, June 1998.

[3] Microtec Research Inc. *VRTX32/86 User's Guide*. Microtec Research Inc., May 1991.

[4] Ivar Jacobson. Re-engineering of old systems to an object-oriented architecture. In *Proceedings of the ACM OOPSLA'92*, pages 340–350. ACM Press, October 1998.

[5] G. Kahn. The semantics of simple language for parallel processing. In *the IFIP Congress 74*, 1974.

[6] Kosmas Karadimitriou, John Tyler, and N. E. Brener. Reverse engineering and reengineering of a large serial system into a distributed-parallel version. In *Proceedings of the 1995 ACM Symposium on Applied Computing*, pages 191–197. ACM Press, February 1995.

[7] N. Kim, M. Ryu, S. Hong, and H. Shin. Experimental assessment of the period calibration method: A case study. *The Journal of Real-Time Systems*, 17(1):41–64, July 1999. To appear.

[8] E. Lee and T. Parks. Dataflow process networks. *IEEE Proceedings*, 83(5):773–801, May 1995.

[9] S. Lim, Y. Bae, C. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, S. Moon, and C. Kim. An accurate worst case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[10] David G. Luenberger. *Linear and Nonlinear Programming*. Addison Wesley, 1989.

[11] V. K. Madisetti, Y. K. Jung, M. H. Khan, J. Kim, and T. Finnessy. Reengineering legacy embedded systems. *IEEE Design and Test of Computers*, 16(2):38–47, April-June 1999.

[12] S. Nadjm-Tehrani and J.-E. Stromberg. Proving dynamic properties in an aerospace application. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 2–10, December 1995.

[13] M. Ryu, S. Hong, and M. Saksena. Streamlining real-time controller design: From performance specifications to end-to-end timing constraints. In *Proceedings of Real-Time Applications and Technology Symposium*, pages 192–203, June 1997.

[14] M. Ryu, J. Park, K. Kim, Y. Seo, and S. Hong. Performance re-engineering of embedded real-time systems. In *ACM Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 80–86. ACM Press, May 1999.

[15] William L. Scherlis. Small-scale structural reengineering of software. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 116–120. ACM Press, October 1996.

[16] Perdita Stevens and Rob Pooley. Systems reengineering patterns. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 17–23. ACM Press, November 1998.

[17] Wind River Systems. The next generation of embedded development tools. *A Wind River Systems White Paper*, 1998.

[18] R. R. Tummala and V. K. Madisetti. System on chip or system on package. *IEEE Design and Test of Computers*, 16(2):48–56, April-June 1999.

[19] T.-Y. Yen and W. Wolf. Performance estimation for real-time distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):1125–1136, November 1998.