

Chapter 2

Computational Capacity-Based Codesign of Computer Systems

David J. Kuck

Abstract This paper proposes a fast, novel approach for the HW/SW codesign of computer systems based on a computational capacity model. System node bandwidths and bandwidths used by the SW load underlie three sets of linear equations: a model system representing a load running on a computer, a design equation and objective function with goals as inputs, and a capacity sensitivity equation. These are augmented with nonlinear techniques to analyze multirate HW nodes as well as to synthesize system nodes when codesign goals exceed feasible engineering HW choices. Solving the equations rapidly finds the optimal costs of a broad class of architectures for a given computational load. The performance of each component can be determined globally and for each computational phase. The ideas are developed theoretically and illustrated by numerical examples plus results produced by a prototype CAPE tool implementation.

2.1 Introduction

System performance is dominated by the performance of individual system components, and balanced component use in a computation. Designers of computer systems and system HW/SW components must face potential system performance instabilities due to component nonlinear performance behavior and imbalanced use. Instability appears in two forms: a given *system* yields widely varying performances over program types, or a given *program* runs at widely varying performances across (similar) system types. Both are common phenomena.

Traditionally, systems that are more stable and productive arise from application-specialization of system components and architectures. Bandwidth metrics can be used to characterize computations by their dominant constituent phases. Matching system HW/SW components to dominant phases for given sets of computations can increase stability. This can be achieved for any program by decreasing that program's performance deviation from the stable value expected for a target system architecture.

D.J. Kuck (✉)
Intel Corporation, Urbana, USA
e-mail: david.kuck@intel.com

This leads to specialization in the marketplace—from embedded processors to GPUs and HPC systems. As computer usage broadens, future designs will continue moving toward more-specialized chips and SW that can exploit them effectively in key markets. Computer system designers and application SW providers need tools and analyses to help them design highly productive systems. Success will depend upon developing methods that can approximate the analysis of future-oriented applications to drive the HW/SW co-design of products.

Significant changes are needed in several areas to meet these needs:

Future-oriented application workloads must be used in computer system design. These may include libraries, reference platform application implementations, or whole-application prototypes, plus the full range of data and usage scenarios.

Leading application development tools must drive the HW design to get top performance and avoid regressions. Tools, compilers, and libraries must be available in advance to allow SW developers to provide the applications above.

System and SW codesign process must allow architects to know deliverable design performance in future markets. Beyond the above, this requires a fast, accurate, mixed fidelity, and comprehensive codesign process.

The performance of a communication network, traffic system, or conference room depends on intrinsic physical characteristics as well as type of load and external ambient factors. For each, peak performance can be defined, but in these examples, noise, weather, and event type, respectively, affect usable *capacity*, in practice. This notion, and the term *capacity*, have been used in several ways relative to computer systems. This paper gives the term a precise, comprehensive meaning.

The paper's contributions are a linear capacity-based codesign process and model (Sects. 2.2, 2.3), methods of formulating and optimizing codesign equations (Sects. 2.4, 2.5), initial numerical codesign results (Sect. 2.6), and an overview of multirate nodes and nonlinearities (Sect. 2.7). Computer HW bandwidth, system architecture, and applications load are all captured in the methodology described. SW performance tuning is reviewed (Sect. 2.8).

2.1.1 Background: Performance Basics

Four main contributors determine computer system performance for a given computation: hardware, architecture, system software, and the application code (including data sets) being run, expressed as in formula (2.1):

$$perf(computation(hw, arch, sw, code)) \quad (2.1)$$

Performance can be expressed as the time consumed or as a rate delivered in running a computation. We will discuss several performance metrics; specific engineering details dictate which are used in a particular codesign effort.

- *Computational capacity* C [4, 5] is a fractional-use metric, ranging between 0 and a maximum value (defined here), and it spans one or more HW components. It is

a joint property of the HW and the computation being performed. For simplicity, we refer to it as *capacity*. At the *computer control level*, capacity is expressed as Eq. (2.2).

$$C \text{ [instructions/sec]} = \frac{\text{clock frequency [clock cycles/sec]}}{CPI \text{ [clock cycles/instruction]}} \quad (2.2)$$

When operations are a more natural metric than *instructions*, we use *operations* O , defined as processing a fixed number of bits. On other occasions, e.g. when data paths are under discussion, capacity measured as BW used in *[bits/sec]* or *[words/sec]* is most natural and this will be used throughout the paper.

- *Bandwidth (BW)* is the standard rate metric for HW design. We use BW to describe individual HW nodes in codesign problems, and as a surrogate for *cost*. HW nodes have a path width measured in *[bits]*, and a delay measured in *[sec]*. BW is expressed in Eq. (2.3).

$$B = \frac{\text{path width [bits]}}{\text{delay [sec]}} \quad (2.3)$$

- *BW used* by a single HW node is written B^u [bits/sec] as one capacity definition. Also, B^u [ops or inst/sec] is required by specific codesign problems. Using Eq. (2.2), assuming $CPI_{\min} = 1$, i.e. one instruction issued per clock,

$$B = B_{\max}^u = \frac{\text{clock frequency}}{CPI_{\min}} = \text{clock frequency [inst/sec]} \quad (2.4)$$

- *Time used* in running a computation as defined with performance *[sec]* units as Eq. (2.5), is particularly useful to compare several computer systems running one *code*. In the form $O = T^u C$, constant load O is forced through the knothole of usable B in used-time T^u . Thus O is a pivot point for time and capacity. Expressions of T are *time-domain* (Eq. (2.5)) and of B or C are *bandwidth- or capacity-domain* (Eqs. (2.2), (2.3)) performance views, respectively.

$$T^u = \frac{O}{C} \quad (2.5)$$

$$C \text{ [operations or bits/sec]} = \frac{O}{T^u} \quad (2.6)$$

- *Efficiency* is the ratio of delivered performance of one or more HW components to its best possible performance, for a given computation defined in the BW domain. Efficiency Eq. (2.7) is usually discussed globally, but Eq. (2.8) expresses a single node x . Their numerators are used as *perf* functions, so Eqs. (2.7) and (2.8) are *perf/cost* surrogates.

$$E = \frac{C}{B} \leq 1 \quad (2.7)$$

$$E_x = \frac{B_x^u}{B_x} \quad (2.8)$$

- *Bandwidth* wasted is an important diagnostic variable, and can be expressed as Eq. (2.9). Combining Eqs. (2.5), (2.7) and (2.9) for a uniprocessor gives Eq. (2.10), where T^{\min} is the time the computation requires with $B^{\text{waste}} = 0$. Equation (2.10) shows that to minimize execution time, for constant O and B , B^{waste} must be minimized.

$$B^{\text{waste}} = B - C = B - EB = B(1 - E) \quad (2.9)$$

$$T^u = \frac{O}{EB} = \frac{T^{\min}}{E} = \frac{O}{B - B^{\text{waste}}} \quad (2.10)$$

- *Balance* refers to a component pair working together perfectly on a computation, and *imbalance* to one component slowing the other. Imbalanced computation is sometimes referred to as *component-saturated* or *component-bound* computation. Balance is closely related to capacity (see Definition 2.1, Sect. 2.3.1).

Generally, capacity and BW-domain analysis are useful for machine design insight, and time-domain analysis is useful for comparing computational performance. Throughout, we assume fixed O across changes to architectures and compilers—this is an idealization, especially for parallelism which often leads to redundancy.

2.1.2 Performance Background Summary

Increasing the system clock frequency may boost a computation's performance directly (unless e.g. latency limits BW), for a fixed architecture. For a constant clock, architecture, BW, SW, or a code's data set affects performance. An important premise for the following is that an initial computer system design is in place for detailed analysis and improvement. The paper discusses methods for improving various bottlenecked parts of a given HW/SW system.

2.2 Codesign Process

Codesign has been used by computer designers with many meanings. We have two symmetrical reasons for using the term: our overall *goal* is to select HW component speeds, compiler and application structures, and to synthesize specific architectural components. Also, we *drive* the process using comprehensive measurements of applications running on existing architectures. We use global HW and SW data together, to improve both HW and SW in the codesign process.

The goals of codesign using global architecture and SW measurements are to avoid design instabilities and regressions by exploring all important aspects of the design space in advance. The codesign process begins with an existing architecture or design, and existing/proposed applications, to produce *optimal* designs. However, when major performance increments are required to meet design goals, architectural synthesis steps are introduced (Sect. 2.7).

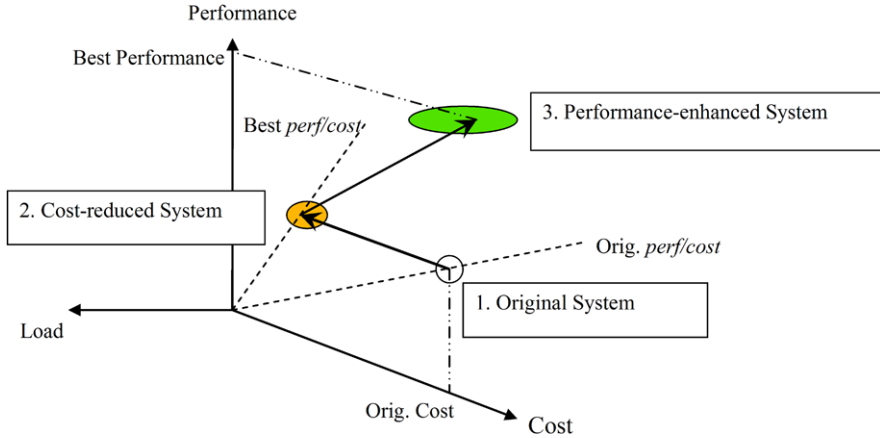


Fig. 2.1 Load normalized to cost-performance plane

2.2.1 Three Dimensions of Codesign

Performance, cost and load are the three dimensions considered in capacity-based codesign. The *codesign process* can accept each as input, and produce each as output; parameters not specified as input are computed as process output.

Performance: As input, a designer chooses specific overall goals or bounds on the performance of specific nodes. The codesign process computes the performance for nodes with unspecified goals.

Cost: A designer can choose specific overall cost goals or bounds for particular nodes; the process computes costs for all other nodes. Cost units can be defined flexibly (to include power, \$, etc. as above); here total BW is a cost surrogate.

Load: This is determined by expected market usage of each application and may be hard to specify. The codesign process can reflect variation in BW used per code type (codelet, see Sect. 2.2.2.1) and weights of percent codelet usage. As output, the process can describe performance and cost ranges, when driven by inputs of load uncertainty.

These three dimensions are considered simultaneously here, not incrementally as in current practice. The codesign process we describe cannot be fully automated. Ultimately, designer interaction is integral to the codesign process, e.g. feasible engineering choices for HW nodes and BW values. The key contribution of this approach, however, is to automate far more of the design process than is currently possible without the capacity-based model.

The three codesign dimensions are shown in Fig. 2.1, which illustrates several important points in the space. Suppose an existing design lies at point 1 in Fig. 2.1. We position the origin of the performance/cost plane in codesign 3-space at the SW load's initial measurement. If the load is constant, cost-performance tradeoffs can be made in the plane shown. Two important codesign transitions within that plane are shown as points 2 and 3.

For any given system and load, the question of reducing its cost arises. In general, it is possible to maintain performance and move horizontally in the cost-reduction direction to the minimum *perf/cost* contour shown passing through point 2. Round point 1 is shown widened a bit, expressing the desirability of taking this step while also allowing for a wider application range. This widening is done by shifting phase weights to represent alternate paths through programs or shifts in usage fractions of various applications, and varying BW used, as may happen with mobile devices that operate over a range of data input BWs. Finally, shifting from point 2 to point 3 enhances performance to a *perf/cost* contour between points 1 and 2 and further enhances the load with a widened point 3.

The progression of Fig. 2.1 continues with shifts off the original load plane. For example, to transitions within the original *perf/cost plane*, but to enhanced loads, or to a completely different *perf/cost plane* for a family of market-focused systems. A client microprocessor may be applicable to a family of hand-held devices with new application loads, but only after re-engineering by changing design BW goals as well as load characteristics—i.e. using new sets of codelets in the codesign process.

2.2.2 Models

The models used for codesign must cover the complete range of HW and SW uses. Each codesign problem considers a system of linked HW nodes (a HW connection exists from each node to at least one other system node); similarly we consider directly or indirectly linked SW units. A single application is assumed to be control/data dependence linked. To include multiprogrammed systems, applications that share HW concurrently can be regarded as indirectly linked.

2.2.2.1 SW Models

SW *load* on a system will be represented by a collection of computational phases that exhibit steady-state B'' on each HW node. Transient B'' behavior is captured by phase transitions. We represent phases by canonical patterns called *codelets*. The two codelet parameters of interest are B'' and weights representing percent-used of total computation time. Ideally, this method will be used to represent every important computation in the design space of interest, in contrast to the benchmark suites frequently used as simplifications. We define the key SW terms:

Codelet: The term *codelet* is used to represent a small, parameterized segment of code that has properties useful in codesign. Ideal codelets will be discussed here, but the codesign process can proceed using only approximations of the ideal, as will be discussed later. The characteristics defining codelets and their use are:

1. Each codelet has approximately *steady-state* B'' , constant B'' provides ideal codesign results

2. Adequate *dynamic coverage* of important applications
3. Maximum *codelet reuse* across applications—minimal number of codelet classes stored in repository
4. Codelets are *compiler recognizable* and achieve *optimized performance* via compilation—codelet size is small enough to analyze automatically, but large enough for maximal system performance.

Phase: A *phase* is a sequence of executed instructions with relatively *uniform B^u* throughout each phase executed for each modeled system component. In practice, a phase will always be measured dynamically in 100s to 1000s of instructions, as *B^u uniformity* cannot be observed at the finest granularity. A single phase may represent a single simple algorithm in a monoprogrammed system, or several simple algorithms in a multiprogrammed system.

Computation: A *computation* is a sequence of phases. Any idle time encountered in a running computer is ignored.

SW Modeling Objective: The objective of *SW modeling* is to map phases discovered in real application computations to codelets in a codelet repository used with a range of data sets per codelet.

The codesign process does not depend critically on finding ideal codelets, but the accuracy and optimality of results will erode as the load is represented by cruder approximations of ideal codelets. For example, a special-purpose design for a particular algorithm whose computation consists of a single phase simplifies the codesign process; highly efficient designs are possible in this case (Sect. 2.4.4). When computations include more phases/codelets, *phase transitions* from one phase to another tend to cause performance sensitivity and can lead to instabilities (Sect. 2.6.2).

2.2.2.2 Computer System HW Models

HW is represented by *nodes* with peak BW values; this can be extended beyond BW to include power consumption, physical area, \$ cost, and so on. *Fidelity* [2] in a given system refers to the node resolution on which a designer chooses to focus, e.g. at a high level one could choose a microprocessor, memory, bus, and network as nodes, and combine these four nodes with a large set of nodes representing individual disks and other I/O devices and controllers. The nodes can be chosen at any fidelity and nodes of various fidelities may be used in one model.

We represent a computer system *architecture* by a *graph* consisting of nodes denoting computer system HW components chosen at any fidelity level, connected by *arcs*, denoting only graph connectivity, i.e. arcs have infinite BW, zero-delay and bidirectional capability. Multiple arcs incident to a node are multiplexed/demultiplexed to and from the node. The following are several types of *nodes*.

Linear nodes: Linear nodes correspond to most low-level system components (e.g. register or arithmetic op). As in Eq. (2.3), their path width is measured in bits and delay is measured in time. Node x has *linear behavior* if its BW is constant and its capacity is a linear function of other node capacities (Eqs. (2.29a), (2.29b)).

Nonlinear nodes: As the modeling abstraction level rises, to vector processors, caches or multicore chips, modeling is harder. A *nonlinear* node has a BW that is a nonlinear function of its linear sub-nodes' capacities. Nonlinear node BW depends on the computation being done, as well as the point at which performance is measured, see [3].

Latency: Since *latency* is delay, the denominator of Eq. (2.3) can be used as a latency surrogate. HW components with BW and latency design issues are represented as two nodes. A memory unit with BW determined by read/write (r/w) cycle time, whose latency depends on wire length (e.g. off-chip delay), is represented by a memory BW supernode (Sect. 2.7) containing two linear nodes: latency and r/w BW. The total delay in Eq. (2.3) is $t_{\text{lat}} + t_{\text{r/w}}$. For word size w , we have Eqs. (2.11), and (2.12). B_{mem} is not a linear function of its constituent BWs, latency or r/w time. In a plot of C_{mem} vs. $C_{\text{r/w}}$, for a given $B_{\text{r/w}}$, varying latency (e.g. disk rotation time) defines a C_{mem} family of load-based B_{mem} values.

$$B_{\text{mem}} = \frac{w}{t_{\text{lat}} + t_{\text{r/w}}} \quad (2.11)$$

$$B_{\text{mem}}^{-1} = B_{\text{lat}}^{-1} + B_{\text{r/w}}^{-1} \quad (2.12)$$

2.2.3 Model Philosophy and Codesign Realities

Several realities separate linear analysis and linear performance response from the real design world. These range from the nonlinearities of HW nodes, through SW variations, to measurement issues, as discussed throughout the paper.

The flow of performance data ranges from continuous (server workloads) to irregularly discrete (laptop use: computation bursts commingled with idle time), and data measurement qualities range from nearly ideal (simulator probes) to crude (Microsoft process-level tools). But if we regard the modeling process as a linear scaling of input data, then even crude approximations can be effectively scaled, if an inverse process exists to carry computed design results faithfully back to the input space. In general, *virtual* HW nodes may be used, if based on appropriate measurement and analysis [2]. For example, a page fault node can be used if there is a way of translating results to and from memory system design (i.e. by virtualizing and devirtualizing the instruction stream measurements).

Figure 2.2 outlines the process of system modeling and obtaining a capacity-based design. The two parameters needed for linear modeling are BW and capacity for each linear node in the model. The details of determining node BW may vary (theoretical peak, microbenchmarking, etc.) but as long as we interpret the results similarly, the method doesn't depend on specific choices. Also, capacity must first be measured and then interpreted for resulting designs.

Mapping is straightforward for linear nodes, following Sects. 2.3, 2.4, and 2.5, and the results are easily mapped back to real hardware. Nonlinear nodes can be dealt with using nonlinear models.

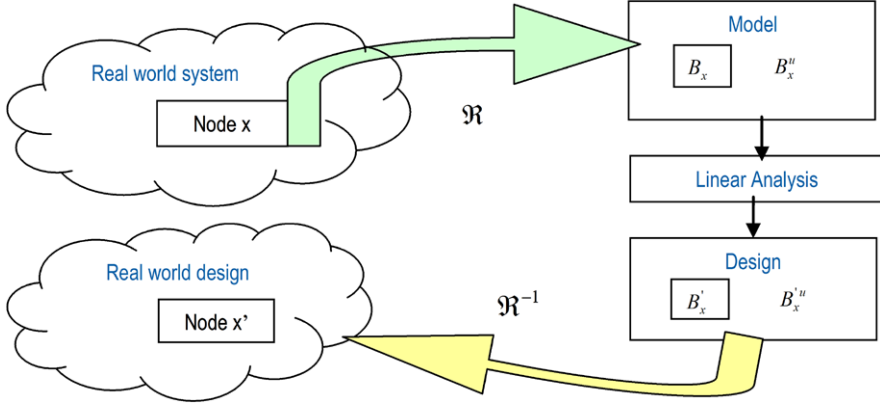


Fig. 2.2 Capacity codesign process

2.3 Linear Computational Capacity Theory

Capacity analysis can be defined on a per computation basis across a system: some simple examples follow.

2.3.1 General Equations for Single Phase Computations

2.3.1.1 Capacity Definitions

For a given system, we define the *computational capacity* of a node pair $\langle x, y \rangle$ as its effective processing bandwidth (BW), either x - or y -bound. The physical definitions of Sect. 2.1.1 will not be detailed further, as specific engineering abstraction and measurement of the real world is required for each codesign problem. The BWs of system nodes x and y are represented by B_x and B_y , respectively, according to some abstraction of the system. B_x and B_y represent amounts of those BWs actually used in a computation, by nodes x and y , respectively, giving Eqs. (2.13).

$$0 < B_x^u \leq B_x \quad \text{and} \quad 0 < B_y^u \leq B_y \quad (2.13)$$

We define the x - y BW ratio in Eq. (2.14) and the x - y used-BW ratio for a given steady-state computation in Eq. (2.15)

$$\alpha_{x,y} = \frac{B_y}{B_x} \quad (2.14)$$

$$\mu_{x,y} = \frac{B_y^u}{B_x^u} = \frac{1}{\mu_{y,x}} \quad (2.15)$$

Definition 2.1 (Node Saturation & Balance) Any node x is *saturated* by a computation if $B_x^u = B_x$. A pair of nodes $\langle x, y \rangle$ is *balanced* if both are saturated, which implies that $\mu_{x,y} = \alpha_{x,y}$.

The following holds for any system under a wide range of conditions (e.g. nodes reflect complete connected HW configuration, continuous operation, no deadlock, etc.), which we assume throughout.

Assumption 1 A running codelet saturates one or more nodes of a given system at each time step.

2.3.1.2 Two Node Systems

Without loss of generality, we analyze the two node system from the point of view of node x .

Definition 2.2 (Computational Capacity with Saturated Node) The *computational capacity of node x* is defined as Eq. (2.16), so from the above we have Eq. (2.17).

$$C_x = B_x^u \quad (2.16)$$

$$C_x = \begin{cases} B_x & \text{if } B_x^u = B_x \quad \text{node } x \text{ saturated} \\ B_x^u & \text{if } B_x^u < B_x \quad \text{node } x \text{ unsaturated} \end{cases} \quad (2.17)$$

In terms of the activity on node y , the second case can be rewritten assuming that node y is saturated, as Eq. (2.18), which we define as the *computational capacity of node x relative to saturated node y* , Eq. (2.19). Because B_x and B_y are defined as non-zero (Eq. (2.13)), capacity is defined only for a pair of nodes that are both actually used a computation. Summarizing, for a pair of nodes, at least one of which is saturated,

$$B_x^u = \left(\frac{\alpha_{x,y}}{\mu_{x,y}} \right) B_x = \mu_{y,x} B_y, \quad \text{for } B_y^u = B_y \quad (2.18)$$

$$C_x = \mu_{y,x} B_y = B_x^u \quad (2.19)$$

Saturated Node Capacity
$C_x = \begin{cases} B_x & \text{if } B_x^u = B_x \text{ and } (B_y^u = B_y \text{ or } B_y^u < B_y), \text{ i.e. } \alpha_{x,y} \geq \mu_{x,y} \\ \alpha_{x,y} B_x / \mu_{x,y} = \mu_{y,x} B_y & \text{if } B_x^u < B_x \text{ and } B_y^u = B_y, \text{ i.e. } \alpha_{x,y} < \mu_{x,y} \end{cases}$

(2.20)

2.3.1.3 Greater than Two Node Systems

Systems of more than two nodes may lead to neither node x nor node y being saturated, unlike the above discussion. This can happen because a system need only have a single saturated node (Assumption 1). Analysis of a multinode system can be built from node-pair analysis, and if one of a pair is saturated, the analysis proceeds as above. Otherwise, a pair of nodes x and y , neither of which is saturated, gives Eq. (2.21). In this case, it is impossible to bound the $\alpha_{x,y}/\mu_{x,y}$ ratio relative to 1 as for the saturated node case in Eq. (2.20). Following Definition 2.2, and since node x is

$$B_x^u < B_x, \quad \text{and} \quad B_y^u < B_y \quad (2.21)$$

$$C_x = B_x^u \quad \text{for} \quad B_x^u < B_x \quad (2.22)$$

unsaturated (Eq. (2.21)) we write Eq. (2.22). Furthermore, expanding Eq. (2.15),

$$B_x^u = \frac{B_x^u}{B_y^u} B_y^u = \mu_{y,x} \quad \text{for} \quad B_y^u < B_y \quad (2.23)$$

Combining Eqs. (2.22) and (2.23), analogously to Eq. (2.19), we have

Definition 2.3 (Computational Capacity with No Saturated Node) The *computational capacity of unsaturated node x relative to unsaturated node y* is defined as

$$C_x = B_x^u = \mu_{y,x} B_y^u \quad \text{for neither node saturated.} \quad (2.24)$$

Combining (2.22) and (2.24) we have Eq. (2.25).

Unsaturated Node Capacity

$$C_x = \begin{cases} B_x^u & \text{if } B_x^u < B_x \\ \mu_{y,x} B_y^u & \text{if } B_x^u < B_x \text{ and } B_y^u < B_y \end{cases} \quad (2.25)$$

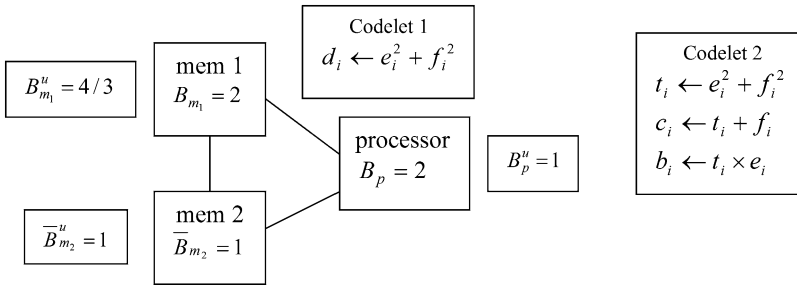
Comparing Eq. (2.20) and (2.25) we see that B s in Eq. (2.20) become B^u s in Eq. (2.25), and the condition $B_y = B_y$ in Eq. (2.20) becomes $B_y < B_y$ in Eq. (2.25). Finally, as nodes x and y saturate, Eqs. (2.20) and (2.25) become identical.

2.3.1.4 General Two-Node Capacity Rule

Combining Eqs. (2.20), (2.22), and (2.25), the general rules are summarized in Table 2.1 (Rule 4 is appended for completeness). At this point the relation between B^u and C can be clarified. Definition 2.2 sets them equal, and in subsequent sections we will generalize the notion of capacity to nodes with multiple connections. However, we use both terms to distinguish contexts. B^u refers to the empirical measurements that are used to define μ (Eq. (2.15)), while C terms are variables used

Table 2.1 Two-node capacity rules

Capacity Equations	Conditions		Rule
	B_x	B_y	
$C_x = \begin{cases} B_x = \mu_{y,x} B_y \\ \mu_{y,x} B_y \\ \mu_{y,x} C_y \end{cases}$	saturated	don't care	1
	unsaturated	saturated	2
	unsaturated	unsaturated	3
$C_y = \mu_{x,y} C_x$	unsaturated	unsaturated	4

**Fig. 2.3** Processor and heterogeneous memory system

to define capacities, (i.e. BW used) in new designs. Finally, μ values are held *invariant* throughout this paper (cf. Sect. 2.5.3). In other words, while B^u values may change when a given program is run on various machines, once a μ value is computed for a computation on any machine, that SW property remains invariant across all machines.

Definition 2.4 (Relative Saturation and Saturation State) The n -node *relative saturation vector* is $\underline{\sigma} = (\sigma_1, \dots, \sigma_n)$ where $0 < \sigma_i = C_i/B_i = E_i \leq 1$. If neither node of the pair $\langle x, y \rangle$ is saturated and $\sigma_x > \sigma_y$, node x is *relatively saturated* to node y , $\sigma_{x,y} = \sigma_y/\sigma_x < 1$. $\underline{\sigma}^s$ is called the *saturation state vector*, where $\sigma_i = 1$ if node i is saturated, and 0 otherwise. $\underline{\sigma}^s$ varies with a design's B values.

2.3.2 Example: Single Processor-Heterogeneous Memory

To illustrate the use of the theory of Sect. 2.3.1, consider the simple machine model in Fig. 2.3, with two memory units and one processor. This could be an abstraction of a system with a register set, memory and arithmetic unit, for example. The nodes are marked with BW values, and the codelets shown are assumed to execute in a loop indexed by i . Architectural assumptions play an important part here, so we sketch those used in this example.

Counting clocks in a cycle-level execution diagram for this model can give B^u values by simply counting cycles used in a periodic instruction pattern. Assume that data are initially stored in mem2. Codelet 1 execution starts by fetching e_1 and f_1 from mem2, and writing them one clock later, respectively, in mem1. When both arguments are in mem1, processing begins with two multiplies followed by an add. The d_1 result is then written back directly to mem2, completing the first iteration of the codelet execution. Other iterations are overlapped, so eventually a steady state execution pattern emerges in a 3-clock cycle.

In practice, these numbers would be collected from running computations on a real system using hardware performance monitoring tools, or could be collected from a simulator for an emerging design. B^u boxes correspond to the system load, codelet 1. The overbar on $\overline{B_{m_2}^u}$ indicates mem2 saturation. If this system were improved by increasing B_{m_2} , mem1 would saturate before the processor because $\sigma_{m_1} > \sigma_p$. Codelet 2 will be discussed in Sect. 2.6.1.

2.4 Single Phase Codesign Equations

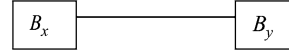
2.4.1 Capacity Equation Generation

For a single codelet, equations may be generated for each arc in a graph to capture all BW and capacity information, but this is not necessary. The $(n - 1)$ arcs in a spanning tree for an n -node graph, is the minimum needed to capture each node in relation to some other node, but no equations need to be written for arcs that close cycles in the graph.

2.4.1.1 Capacity Equation Generation Algorithm

This algorithm suffices to capture the capacity equations for any graph.

- Step 1.* Start with any node and build a spanning tree for the entire graph from that root.
- Step 2.* Write capacity equations for each arc in the spanning tree using Table 2.1. As saturation patterns for a new design are unknown, codesign model equations (Sect. 2.4.3) use only Rule 3 in either orientation. (*Initial equations* can capture the original system and computation by using all three rules, for 2, 1, and 0 nodes saturated, respectively.)
- Step 3.* This generates $n - 1$ equations for a graph of n nodes, as there is one equation per arc in the spanning tree. If we think of capacity as a nodal relation on a spanning tree, *transitive relations* may be formed between non-adjacent nodes in the graph. This idea and Definition 2.4 are useful in analyzing the sensitivity of solutions (as in the example of Sect. 2.3.2) or in expediting specialized solutions for specified nodes.

Fig. 2.4 Two-node graph

2.4.2 Codesign Equations

We assume that μ values are constant, while B and C values are variables, i.e. that we are designing HW (B values) to suit given SW (μ) needs. In Sect. 2.5.3, the design of SW in terms of HW will be discussed briefly.

Physical constraints as well as capacity equations are needed to form a complete design equation set. This section contains examples of linear capacity and physical equations that describe a computer system and computation. The nodes can represent any fidelity, but at a high level one may choose multirate nodes whose behaviors vary, depending on their load. Multirate and nonlinear nodes will be discussed further in Sect. 2.7. For this section, it suffices to assume that nodes are chosen at a level that allows linear performance equations to hold.

Physical Constraints It is generally true that for any node x , $0 < B_x^u \leq B_x$. This follows from obvious physical considerations, and because capacity is defined only for nodes where $B^u > 0$ (Eq. (2.13)). To formulate general equations, we represent unknown B^u by C variables, so we rewrite $0 < C_x \leq B_x$ as two inequalities which must be satisfied in all solutions:

$$B_x - C_x \geq 0 \quad (2.26)$$

$$C_x > 0 \quad (2.27)$$

2.4.3 Single Phase Models and Characteristic Equation

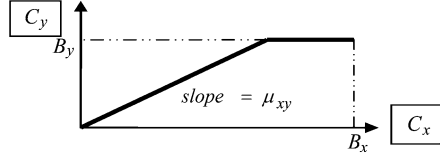
We introduce the form of the general design equations by starting with a single phase running on simple systems.

Two Node Systems Figure 2.4, the two node graph case, with saturated node x , has the initial capacity

$$\mu_{xy} B_x - C_y = 0 \quad (2.28)$$

Equation (2.28) using Rule 2 of Table 2.1 (rewriting $C_{yx} = C_y$). To cover all possible BW values in system designs executing this computation, Fig. 2.4 is represented by either of Eqs. (2.29a), (2.29b) (Rule 3 Table 2.1) as in general, given a constant μ value, either node x or node y may be saturated in a solution for particular B values (Eq. (2.29a) reduces to Eq. (2.28) if node x is saturated). For any two node system with $B_x = B_y$ and $B_x^u = B_y^u$, node x will saturate in a *new design* with $B_x \ll B_y$, while node y will saturate if $B_x \gg B_y$. Equation (2.29a) for constant μ_{xy} defines x as a *linear node* relative to other nodes y . For $B_x > B_y$, Fig. 2.5 shows the

Fig. 2.5 Linear node-pair capacity



behavior of Fig. 2.4, beginning with the slope of Eq. (2.29a) (Rules 3, 4; Table 2.1), then $C_y = B_y$ when B_y saturates (Rule 1; horizontal break), and becomes undefined for $C_x > B_x$ (no Rule). Since $B_x > B_y$, Rule 3 applies until $C_x > B_x$. A spanning tree has $n - 1$ nodes, so Fig. 2.4 has one capacity equation. Following Eqs. (2.26) and (2.27), the *physical equations* are Eqs. (2.30) and (2.31).

$$C_y = \mu_{xy} C_x \quad (2.29a)$$

or

$$C_x = \mu_{yx} C_y \quad (2.29b)$$

$$B_x - C_x \geq 0, \quad \text{and} \quad B_y - C_y \geq 0 \quad (2.30)$$

$$C_x > 0, \quad \text{and} \quad C_y > 0 \quad (2.31)$$

We combine these in the *single-phase model system*, Eq. (2.32) as the product of computational parameter matrix M containing parameters μ , 0 and ± 1 , and *design vector* d , partitioned into b and c . *Capacity vector* c corresponds to the performance of a given design, measured in B , and *bandwidth vector* b represents the cost of obtaining that performance, measured in B . The positions of equality and inequality signs in Eq. (2.32) denote numbers of equalities (starting at =) and inequalities (starting at \leq), and 0 is a zero column.

$$M \underline{d} = M \begin{bmatrix} \underline{b} \\ \underline{c} \end{bmatrix} = \begin{bmatrix} 0 & 0 & \mu_{xy} & -1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} B_x \\ B_y \\ C_x \\ C_y \end{bmatrix} \begin{matrix} = 0 \\ \geq 0 \\ \geq 0 \\ > 0 \end{matrix} \quad (2.32)$$

$$M = \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \\ M_{31} & M_{32} \end{bmatrix} = \begin{bmatrix} 0 & M \\ I & -I \\ 0 & I \end{bmatrix} \quad (2.33)$$

General Systems The above discussion easily generalizes to a system of n nodes. M is a $(3n - 1) \times 2n$ matrix representing the computation. The $2n$ columns correspond to a B and C per node. The $3n - 1$ rows include $n - 1$ capacity equation rows plus $2n$ for physical inequalities, while \underline{b} and \underline{c} are n -element vectors representing bandwidth and capacity, respectively. We can partition M as Eq. (2.33), where M is a matrix of μ , 0 and -1 values, I is the identity, and 0 the null matrix. M_{11} and M_{12} are $(n - 1) \times n$, and the other M_{ij} are $n \times n$ matrices. As each C_i represents one

node's performance, by combining these we can summarize the overall *system performance* as a linear metric where the w_i represent design-importance or emphases on each node's contribution to overall system performance.

$$perf(\text{overall system}) = C_{\text{system}} = \sum_{\text{nodes}} w_i C_i \quad (2.34)$$

From Eq. (2.32), we derive the *single-phase characteristic equation* of an n -node computation, Eq. (2.35), where the inequalities of Eqs. (2.30) and (2.31) have been augmented with one slack variable per physical equation (indicated by primes) to obtain a $(3n - 1) \times 4n$ underdetermined system of equations. Identical graph topologies arising from distinct architectures lead to equations of the same nonzero patterns, but represent distinct architectural behavior via distinct μ values per position.

$$M' \underline{d}' = M' [\underline{b}', \underline{c}']^T = 0 \quad (2.35)$$

The characteristic equation contains complete information about performance and cost for any HW system running the single-phase computation used to generate it. The specifics of each computation are represented by μ values. The solution will have $k \geq 1$ saturated nodes. Myriad real HW systems are described by one characteristic equation, in general. Section 2.5.6, gives codesign optimizations for selecting a few practical candidate system designs.

2.4.4 Observations

Obs. SP1: For a single phase computation, it is always possible to design an n -node system with all nodes saturated.

Obs. SP2: Any one node's performance can be set to an arbitrary goal (Sect. 2.5.2) while maintaining Obs. SP1.

Obs. SP3: In any single-phase computation with some unsaturated nodes, changing the BW of saturated nodes changes system-wide performance; for any unsaturated node x , changing B_x such that $B_x < B_x$ does not affect performance.

Observations SP1 and SP2 show how effectively the codesign process can be carried out for single phase computations. They provide a heuristic justification for the many demonstrations since the beginning of computing history that HW specialized to a single algorithm can be far more cost-effective than general purpose systems. In the future, massively multicore chips could allocate substantial real estate to an extensive set of algorithm-level processors. By Obs. SP3, unsaturated node BWs can float until they all reach saturation in the form of Obs. SP1.

2.5 Multiphase Codesign Equations

2.5.1 Multiphase Model and Characteristic Equations

Adding multiple computational phases does not affect the vector \underline{b} in Eq. (2.32), as the machine BW is defined by one set of nodes used in all phases. However, using the designed HW, each phase generally produces distinct performance characteristics. Thus, if a computation has m phases, the \underline{c} vector becomes m times larger than for the single-phase case. We represent the collection of phase performance vectors in Eq. (2.36), where m phases are represented in Eq. (2.37).

$$M\underline{d} = M[\underline{b}, \underline{c}^{\text{loc}}]^T \quad (2.36)$$

$$\underline{c}^{\text{loc}} = [\underline{b}_1, \dots, \underline{c}_m]^T \quad (2.37)$$

As an example, expanding on Sect. 2.4.3, we show a 2-node *two-phase model system* in Eq. (2.38). Assume that in the second phase the saturation is reversed from Fig. 2.4; y is saturated and x is not. Per phase, this gives one capacity equation; third subscripts denote phase numbers. In the M_{phy} partition of M (Eq. (2.38)), rows 3–6 are physical equations of the form $B - C \geq 0$ (Eq. (2.30)); the next four rows are an identity matrix corresponding to Eq. (2.31) for the two phases. There are n elements in \underline{b} and in each \underline{c} vector, for a total of $n(m + 1)$ columns in M and rows in \underline{d} .

$$M\underline{d} = \begin{bmatrix} M_{C^{\text{loc}}} \\ M_{\text{phy}} \\ M_{C^{\text{glob}}} \end{bmatrix} \begin{bmatrix} \underline{b} \\ \underline{c}^{\text{loc}} \\ \underline{c}^{\text{glob}} \end{bmatrix} = \begin{bmatrix} 0 & 0 & \mu_{xy,1} & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & \mu_{yx,2} & 0 & 0 \\ 1 & & -1 & & & & 0 & 0 \\ & 1 & & & & -1 & 0 & 0 \\ 1 & & & & -1 & & 0 & 0 \\ & 1 & & -1 & & & 0 & 0 \\ & & 1 & & & & 0 & 0 \\ & & & 1 & & & 0 & 0 \\ & & & & 1 & & 0 & 0 \\ & & & & & 1 & 0 & 0 \\ 0 & 0 & \phi_1 & 0 & \phi_2 & 0 & -1 & 0 \\ 0 & 0 & 0 & \phi_1 & 0 & \phi_2 & 0 & -1 \end{bmatrix} \begin{bmatrix} B_x \\ B_y \\ C_{x,1} \\ C_{y,1} \\ C_{x,2} \\ C_{y,2} \\ C_x^{\text{glob}} \\ C_y^{\text{glob}} \end{bmatrix} \begin{matrix} = 0 \\ \geq \underline{0} \\ > \underline{0} \\ = \underline{0} \end{matrix} \quad (2.38)$$

In general, we will write a block of equations as above for each of m phases, so from Eq. (2.32), M is an $m(3n - 1) \times (m + 1)n$ matrix. Some nodes may not be used in a particular phase, e.g. no disk accesses are made, so, as capacity C_{xy} is undefined for unused node y (Eq. (2.13)), it may be dropped from the equation set for that phase. In solving such systems, we can drop nodes for phases whose use approaches the machine's zero value. This leads to a linear system where each

phase is reduced in size to represent those nodes active per phase. Generalizing Sect. 2.4.3 with slack variables leads to an underdetermined system, as each phase is underdetermined.

Global component performance in the multiphase case is determined by contributions from each phase. Each phase has a weight ϕ_j , $1 \leq j \leq m$, defined by some combination of the running time of an application segment, the importance of an application containing the segment, etc. The *multiphase global performance* of a HW node x is Eq. (2.39). The last two global performance rows of the model system Eq. (2.38) correspond to Eq. (2.39), for each node, x and y . For an n -node system, this adds n more rows and columns to Eq. (2.38), so M is an $[m(3n - 1) + n] \times [(m + 2)n]$ matrix. Equation (2.38) yields all BW and performance values for a computation on a computer system, given the μ ratios and ϕ weights.

$$\text{global perf}(\text{node } x) = C_x^{\text{glob}} = \sum_{\text{phases } j} \phi_j C_{x,j} \quad (2.39)$$

The ϕ_j phase-weights are functions of node BW in general, because running times of individual phases may vary relative to each other based on specific node BWs. Boosting the BW during the design process of node x , unsaturated in phase k , can reduce the running time of phases for nodes that were B_x -bound. In principle, we should readjust the $\phi_j(x)$ values for all nodes (or the most sensitive nodes). As the sizes of changes may be small, and ϕ_j also depends on other qualitative weights, we avoid the complexity of varying ϕ_j , which could be done iteratively.

For multiphase systems, we expand the characteristic equation (Eq. (2.35)) to include global performance equations, and augmenting Eq. (2.38) with $2mn$ slack variables yields Eq. (2.40), an underdetermined *multiphase characteristic equation*, where M is a $[(3m + 1)n - m] \times (3m + 2)n$ matrix.

$$M' \underline{d}' = M' [\underline{b}', \underline{c}'^{\text{loc}}, \underline{c}'^{\text{glob}}]^T = 0 \quad (2.40)$$

2.5.2 Codesign Equation

For system design, we rewrite Eq. (2.38) as the *multiphase codesign equation*, Eq. (2.41). Chosen coefficients of \underline{b} and $\underline{c}^{\text{glob}}$ are removed from M and \underline{d} , and corresponding positions are set to constant goal values in \underline{g} with appropriate sign changes. The codesign equation can be used to set any BW, capacity or weight goals a designer chooses to target. This reduces the number of columns of M and the size of \underline{d} by g , the number of elements moved to \underline{g} . A codesign study may select $\leq n$ global performance equations of the form Eq. (2.39), so $[m(3n - 1) + p] \times [(m + 2)n - g]$, $p \leq n$ is the size of M . In a codesign study, the \underline{g} values may be chosen to sweep out regions of the overall design space to find optimal designs. See Sect. 2.6.

$$\boxed{M \underline{d} = M \begin{bmatrix} \underline{b} \\ \underline{c}^{\text{loc}} \\ \underline{c}^{\text{glob}} \end{bmatrix} = \begin{bmatrix} \underline{b}^{\text{goal}} \\ 0 \\ \underline{c}^{\text{goal}} \end{bmatrix} = \underline{g}} \quad (2.41)$$

2.5.3 Software Design Equations

Although it will not be explored in any detail here, the model Eq. (2.38) has a second interpretation. This paper assumes BWs are variables and μ values are constants. Inverting these relationships, assume a fixed HW system, and instead of Eq. (2.38), write Eq. (2.42), where B is a matrix of BW values and $\underline{\mu}$ is a vector of measured or unknown μ values. Setting capacity and μ goals allows the computation of C and μ values, by reasoning analogous to the ideas of this paper. A key difference of SW tuning for given HW, from HW design is that via $\underline{c}^{\text{loc}}$, each phase can be tuned to achieve desired C and μ values, allowing more degrees of freedom than choosing node BW values. It can be viewed as a way of tuning codelets to specific applications. Substantial tuning work may be involved, but performance targets are guaranteed if it succeeds. Some performance tuning methods related to the equations of this paper (ranging from basic capacity to sensitivity analysis) are outlined in Sect. 2.8.

$$B\underline{d} = B[\underline{\mu}, \underline{c}^{\text{loc}}, \underline{c}^{\text{glob}}]^T \quad (2.42)$$

2.5.4 Multiphase Performance Observations

Obs. MP1: Generally, no system can have all nodes remain saturated throughout a multiphase computation.

Obs. MP2: For a given computation on a well-designed system, each phase saturates some node(s); and each node will be saturated by at least one phase.

Obs. MP3: Linearly scaling all linear model BWs by factor a scales all phase and global capacities by a factor of a .

These observations raise a question. Since we cannot achieve saturation of all nodes throughout a multiphase computation, how should a good design be defined? As discussed earlier, the *benefits* of a design are represented by capacities, and the *costs* of obtaining capacities are BWs. A desirable goal is minimizing *BW wasted per node*, $B_i^{\text{waste}} = (B_i - C_i)$, summed across all nodes. It is exactly the achievement of all-node saturation that is made by optimized single phase computations, Obs. SP1. From Eq. (2.10) minimizing BW waste is equivalent to minimizing time, overall.

2.5.5 Overall System Optimization

In the multiphase case, using Eq. (2.39) the overall system *performance* corresponding to Eq. (2.34) is Eq. (2.43). For n nodes and m phases, the overall system *cost* is Eq. (2.44), so the objective of minimizing total *wasted BW*, B^{waste} , Eq. (2.45), is the difference between Eqs. (2.43) and (2.44). In a codesign problem, certain BW

values and performance goals may be chosen a priori, using Eq. (2.45) to minimize the remaining unknown *cost* (BW) values.

$$perf \text{ (overall system)} = C_{\text{system}} = \sum_{\text{nodes}} w_i \sum_{\text{phases}} \phi_j C_{i,j} \quad (2.43)$$

$$cost \text{ (overall system)} = B_{\text{system}} = \sum_{\text{nodes}} B_i \quad (2.44)$$

The system codesign problem formulation thus becomes the optimization problem of minimizing an objective function (Eq. (2.45)), subject to a set of linear constraints (Eq. (2.41)), a *linear programming* formulation of system codesign. Evaluating simplex solutions at many design points can lead to nonlinear surfaces in codesign space (see Sect. 2.6).

$$\min B_{\text{system}}^{\text{waste}} = \min \sum_{i=1}^n \left(B_i - w_i \sum_{j=1}^m \phi_j C_{i,j} \right) \quad (2.45)$$

2.5.6 Sensitivity Analysis

Several sensitivities are of interest in codesign studies.

Definition 2.4 (continued) The *n-node m-phase* generalization of relative saturation state (Definition 2.4, Sect. 2.3.1) is the $n \times m$ *saturation matrix* $\Sigma = [\underline{\sigma}_1, \dots, \underline{\sigma}_m] = [C_{i,j}/B_i] = [E_{i,j}] = [\sigma_{i,j}]$, where $\underline{\sigma}_j^T$, $1 \leq j \leq m$ has the form of Definition 2.4. Each set of Σ values defines a computation's *saturation state* Σ^s , where $\sigma_{i,j} = 1$ for saturated node i in phase j , and 0 otherwise. Σ^s varies with phases as well as the B values chosen in each design.

The collective performance- or cost-sensitivity of a given computer system and computation set can be analyzed by examining the saturation matrix. We define *capacity sensitivity*, relating all nodes i to any particular node y in phase j , as Eq. (2.46). Similarly, *bandwidth sensitivity* relating all nodes i to any particular node y in phase j is defined as Eq. (2.47). Using Definition 2.4 for $y = i$, the relation between Eqs. (2.46) and (2.47) is $\Sigma'_B = E \otimes \Sigma'_C$; \otimes is the Hadamard product.

$$\Sigma'_{Cy} = \left[\frac{\partial \sigma_{i,j}}{\partial C_{i,j}} \right] = \left[\frac{\mu_{yi,j}}{B_i} \right] = [\sigma'_{Cy i,j}] \quad (2.46)$$

$$\Sigma'_{By} = \left[\frac{\partial \sigma_{i,j}}{\partial B_i} \right] = [-\mu_{yi,j} C_{y,j} / B_i^2] = [\sigma'_{By i,j}] \quad (2.47)$$

Σ'_B offers a view of the most effective BW changes to reduce a design's relative saturation (or efficiency) sensitivity, across all node BWs in the codesign process.

Σ'_C provides a similar effect in reducing the sensitivity of a key node across the capacity sensitivities of all other nodes. Both could be phase-weighted in practice.

This section shows that capacity sensitivity, Eq. (2.46), a function of the model system, joins the model system and codesign optimization equations as a third key element in understanding the codesign problem.

2.6 Using the Codesign Model

Next we explore some codesign issues by sweeping through parts of the design space for noncontrived, simple 3-node, 2-phase examples. Realistic numbers of nodes and phases would make the picture more complex, but follow similar patterns. Sections 2.6.1 and 2.6.2 concern cost and performance sensitivity, respectively, and performance sensitivity is broken into several cases in Sect. 2.6.2. We define *performance stability* as the ratio of maximum to minimum performance over a collection of computations; when an empirical threshold is exceeded, a system is said to be *unstable*. Potential sources of performance sensitivity and nonlinearity that are discussed here and can lead to instability include variations in saturation state by discrete choices of node BW values, architecture changes, and variations in phase weights. A prototype codesign tool CAPE was used to produce the figures shown.

2.6.1 Cost Reduction

Computer manufacturers build only a discrete set of computer systems, and these are made from a discrete set of subsystems, each of which is built from a discrete set of components. For example, most sizes and speeds of memory along a continuum are not feasible design choices for real memory systems—designers use what is reasonable to fabricate in volume. A codesign tool must be constrained to choose among these engineering options. Linear analyses over ranges of discrete constraint choices create nonlinear codesign surfaces that approximate design realities.

Consider the 3-nodes of Fig. 2.3, plus a second phase codelet 2, which yields $B_{m_2}^u = 17/18$, $\bar{B}_{m_1}^u = 36/18 = 2$, and $B_p^u = 22/18 = 11/9$. In contrast to phase 1, mem1 is saturated here, so $\mu_{m_1, m_2, 2} = B_{m_2}^u / B_{m_1} = 0.944 \ 2 = 0.472$, and $\mu_{m_2, p, 2} = B_p^u / B_{m_2}^u = 1.22/0.944 = 1.29$. Assuming that $\phi_1 = \phi_2 = 0.5$, we can find minimum cost solutions among discrete BW values, satisfying engineering codesign constraints. We define system performance in terms of the processor as $C_p^{\text{glob}} = (C_{p,1} + C_{p,2})/2 = 1.11$. Varying processor and memory BW with a step size of 0.1 to simulate engineering constraints, while maintaining original performance, Table 2.2 shows several cost-reduced solutions relative to the original $B_{\text{system}} = 5$ (Eq. (2.44)). These range from $B_{\text{system}} = 4.1$, an 18% cost reduction, to $B_{\text{system}} = 4.3$. The range of BW options covered may have important engineering consequences. For example, the mem1 and proc BWs are reduced from the original design, while mem2

Table 2.2 Low cost solutions vs. orig. cost = 5

B_{system}	C_p^{glob}		Bandwidths
Cost	m1	m2	p
4.100	1.7000	1.2000	12000
4.200	1.7000	1.2000	13000
4.200	1.8000	1.2000	12000
4.200	1.9000	1.1000	12000
4.300	1.7000	1.2000	14000
4.300	1.8000	1.2000	13000
4.300	1.9000	1.1000	13000
4.300	1.9000	1.2000	12000
4.300	2	1	13000
4.300	2	1.1000	12000

ranges up to 20% higher than the original. From a design flexibility point of view, note that for each component, this approach provides the designer with component BW choices in a 15% to 20% range. Table 2.2 is the type of tool output that brings designers' decision-making into the codesign process.

Variations of the performance goal for a single node can be satisfied by linearly scaling system cost. However, when minimum cost is sought, or two or more node performance goals change independently, the resulting cost surface can become non-linear. For example, the most demanding performance goal can require disproportionate BW for that node compared to other nodes. We refer to this as a *nonlinear* cost function of performance.

2.6.2 Performance Sensitivities and Instabilities

We consider performance (Eq. (2.1)), in Sect. 2.6.2.1 with variable HW/architecture and constant SW/code, and in Sect. 2.6.2.2 with variable SW/code and constant HW/architecture. Exploring the solution space at phase transitions in computations reveals a source of *nonlinear performance* behavior caused by saturation state changes (Sect. 2.5.6). For each HW node i , as a program's $C_{i,j}$ changes across phase j transitions or within phases as data sets vary, the Σ row values change, and as a program is moved from one machine to another, the column values also change. C_p^{glob} , as a function of B_{m_1} and B_{m_2} , exhibits three distinct linear regions in Fig. 2.6, where three planes (breaks at dotted lines) form a performance surface viewed from below (higher is better). The heavy line shows the $B_{m_2} = 0.8$ contour. Discrete choices of node BW values and variations in phase weights are other potential sources of performance sensitivity and instability (in cases of extreme sensitivity).

Figure 2.7 shows a C_p^{glob} vs. B_{m_1} slice through the surface of Fig. 2.6 for $B_{m_2} = 0.775$, cutting across three regions. This value illustrates some difficulties of doing

Fig. 2.6 Processor performance vs. memory BW

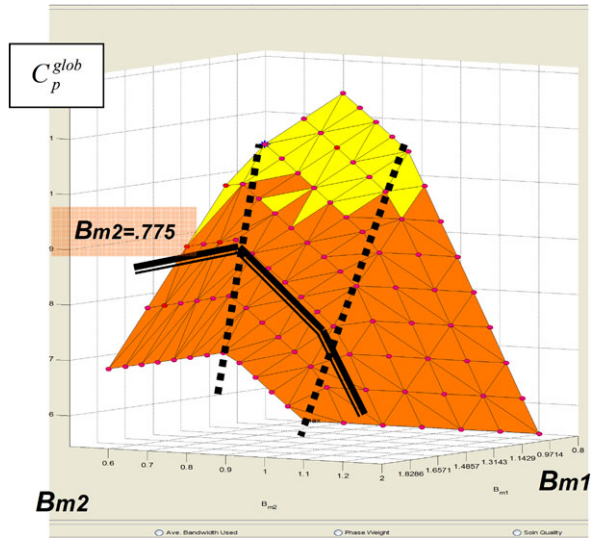
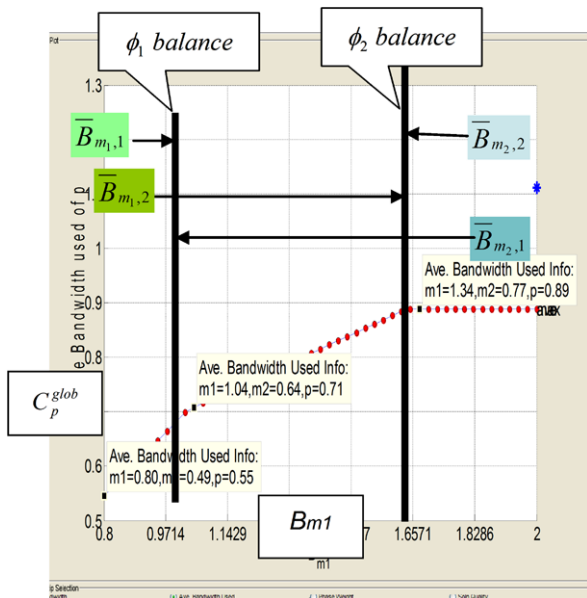


Fig. 2.7 Processor performance vs. B_{m1} , showing perf regions; $B_{m2} = 0.775$



reduced-performance and cost redesign of the original system. The * on the right is the original design point, with $C_p^{glob} = 1.11$. The $\langle m_1, m_2 \rangle$ balance points for each phase can be computed using $\mu_{xy} = \alpha_{xy}$ (Definition 2.1). With $B_{m2} = 0.775$ for phase 1, $\mu_{m_1 m_2} = \alpha_{m_1 m_2}$, so $B_{m1} = B_{m2} / \mu_{m_1 m_2, 1} = 0.775 / 0.75 = 1.033$, and for phase 2, $B_{m1} = 1.64$. These breaks are shown in Fig. 2.7 labeled as balance points

for each of the phases. For $B_p = 1$, they define three saturation states,

$$\Sigma_{\text{left}}^s = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 0 & 0 \end{bmatrix}, \quad \Sigma_{\text{center}}^s = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \text{and} \quad \Sigma_{\text{right}}^s = \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 1 \end{bmatrix},$$

following Definition 2.4, Sect. 2.5.6, with $\underline{\sigma}_j = [p, m_1, m_2]$. $B^{\text{waste}} > 0$ is indicated by 0-rows, which right-state designs offset by B_{m_1} insensitivity.

2.6.2.1 Sensitivity of Performance to the System

Consider the system sensitivity caused by saturation state transitions in moving an application from one system to a similar one. In the leftmost region of Fig. 2.7 (B_{m_1} saturated in both phases), if B_{m_1} values differ on two similar architectures performance will be affected more than in the other two regions due to a linear tradeoff between B_{m_1} and processor performance. In the center region, the performance benefit of incremental B_{m_1} change is about half as great. The rightmost region is insensitive, as C_p is independent of B_{m_1} . Two machines to the right of the ϕ_2 balance point show no performance change as B_{m_1} varies; two machines to the left of the ϕ_1 balance point show a 20% performance variation in the region graphed.

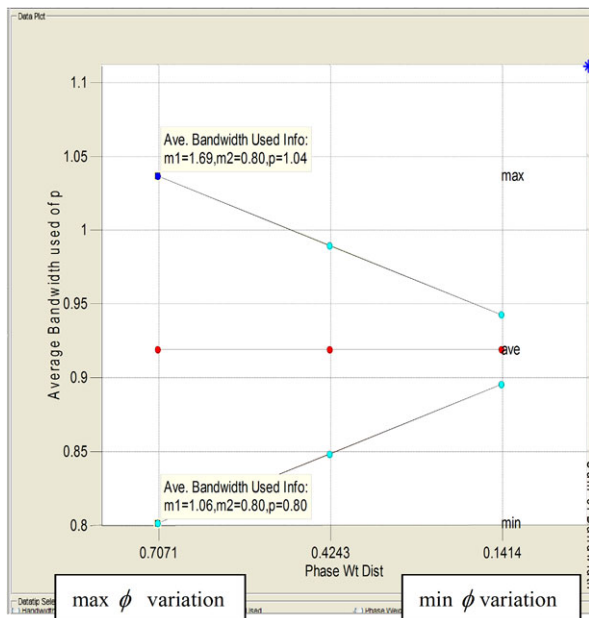
Analyzing this via Eq. (2.46), with B_{m_1} values at phase balance points and μ 's computed from data above gives

$$\begin{aligned} \sigma'_{B_p \ B_{m_1}, \ \phi_1 \ \text{bal}} &= \frac{-\mu_{p \ m_1, 1} C_{p, 1}}{B_{m_1}^2} = \frac{-1.33 \times 0.7}{(1.033)^2} = -0.87 \\ \sigma'_{B_p \ B_{m_1}, \ \phi_2 \ \text{bal}} &= \frac{-\mu_{p m_1, 2} C_{p, 2}}{B_{m_1}^2} = \frac{-1.64 \times 0.89}{(1.64)^2} = -0.54 \end{aligned}$$

The numerical sensitivity at the left balance point exceeds that at the right by a factor of 1.6, and evaluation at the midpoints of the two sloping lines yields a ratio of 1.4, in general agreement with the plot for related concepts. Similar results can be obtained for B_{m_2} sensitivity, corresponding to a slice across Fig. 2.6 along the B_{m_2} axis.

To explain how real-world design efforts might produce systems with performance sensitivity that varies more than necessary, imagine two design teams, one working to the left of the phase 1 balance point, and one to the right. Assume that neither team has a global view of the design space beyond what typical simulation studies allow [9]. The first team will be more easily inclined to increase B_{m_1} based on incremental studies, subject to cost constraints. In competition with other teams working in the leftmost region, under fixed cost budgets, team 1 designers could make bigger design errors, by insufficiently incrementing B_{m_1} , than team 2 or other design teams working to the right of the balance point. In general, operating at a balance point is locally optimal, but without global oversight all design teams are likely to err. Adding more-detailed nodes at sensitive points provides zoom-in on hot spot design.

Fig. 2.8 Phase weight sensitivity



2.6.2.2 Sensitivity of Performance to the SW Load

Future workloads are impossible to know with certainty, but the capacity-based process allows approximating them as changes to current workloads. Varying BW used and codelet weights allows approximation of new paths through existing applications (data set changes), emerging algorithms and applications (codelet variations), and ranges of data rate inputs. Driving the linear analysis with such perturbations also leads to nonlinear performance surfaces. To simulate uncertainty in data-dependent program paths, or application market-importance variation over time, we can vary the phase weights in some range ($UB - LB$), where $LB \leq \phi_1, \phi_2 \leq UB$, $\sum_j \phi_j = 1$, using a constant increment. The choice of UB and LB depend on specific design constraints. This has the effect of varying $C_{i,j}$ in saturation states.

Figure 2.8 plots the combinatorial magnitude of the distribution of phase weights, ranging here from 0 to 1 in steps of 0.2. The maximum variation of C_p^{glob} in this example is 30%—from 0.80 to 1.04. The details are not shown, but as the phase weight range increases, maximum proc performance variation increases: from 9% (not shown for phase weight range 0.3 to 0.7) to 30% (Fig. 2.8). Comparison at the two balance points (Fig. 2.7) shows greater stability at the ϕ_1 balance point; the opposite of the HW stability conclusion (Sect. 2.6.2.1). The explanation of this may depend on the higher peak and average performance values achieved for higher B_{m1} . Thus, *load instability* can be manifest on a single system by running two similar applications, or one application with varying data sets.

2.6.2.3 Performance Instability Conclusions

Explaining performance instability is a complex subject, and one simple example can only provide an illustration. This section demonstrated performance variations in one application under architectural changes in Fig. 2.7, where the left region demonstrates a 20% performance range, and load changes in Fig. 2.8, where 10% to 30% performance variations arise as a function of one computation's phase weight variation on a fixed architecture.

For this simple system, it is easy to demonstrate the basic mechanisms by which instabilities in real computer systems arise. This heuristic discussion proves nothing about instability, but points the way to more analytical methods for finding and evaluating performance instabilities based on BW/architectural change as well as load change.

2.6.3 Architectural Variations Affecting Capacity

Using the methods of this section, a design space can be explored for critical architectural changes. For example, given an accurate laptop model for a comprehensive workload, how would a solid-state disk noticeably improve performance. By reducing disk latency appropriately, the shift from hard drive to SSD could be modeled, and those phases (applications) could be discovered for which delivered processor performance increased significantly. Further, designers could examine the potential of small on-chip RAM supplemented by SSD.

The appropriate model could be driven by C and μ values estimated from the original system—on-chip RAM and SSD latencies would be much reduced, while page faults would increase. The tool would show performance improvement per application together with sensitivities to the C and μ parameter estimates. This could quickly provide a crude view of potential architecture vs. market tradeoffs, together with some sensitivities.

2.7 Multirate Nodes

The two types of multirate node have BWs that vary with computational load.

Supernode: A supernode is any connected set of linear nodes. It can be used to denote a subsystem's variable performance behavior in either HW (e.g. memory latency) or architecture (e.g. queues with variable internal latency).

Nonlinear node: A nonlinear node's BW is a nonlinear function of other nodes' capacities. Examples include a parallel processor, cache hierarchy or vector processing unit.

Multirate nodes arise in two ways:

1. A designer chooses a model fidelity that includes supernodes or nonlinear nodes, forcing their analysis.
2. Capacity analysis yields a BW objective exceeding current technology limits, forcing multirate node synthesis.

Analysis: Most RTL-level component performance responses are linear relative to BW or latency. But when architecturally linked and driven by applications imperfectly matched to the architecture, overall performance response can be nonlinear. Multirate node C and μ values generally depend nonlinearly on HW component size metrics and how the computation interacts with the detailed node structure, e.g. loop vectorization or blocking for cache reuse [2].

Synthesis: When linear analysis applied to performance enhancement of a design calls for a node BW that is infeasible using available linear components, a multirate node may be synthesized. Using performance objectives obtained by the capacity-based solver as the multirate node BW requirement, a secondary method can specify its internal structure, e.g. the required number of cores for a multicore component.

2.8 Related Work

Discussion of compute vs. memory or I/O bound programs, the von Neumann architecture bottleneck, and designing systems to match given applications or algorithms, have driven computer design for 50 years [6]. Obtaining, analyzing and interpreting large volumes of performance data present major obstacles that have been addressed in many ways. Deterministic and stochastic models with sampling from the application level (benchmarks) to the trace level (HW performance counters) followed by various discrete event simulators and statistical models are used in specialized or combined ways. Stochastic methods tend to work well for steady-state computations, while discrete event simulation handles all situations but much more slowly. Multiple system types have evolved to cover multiple market needs.

Capacity-based codesign can handle at linear programming speeds, both steady-state and transient system behavior. Codelet coverage is the key need; it can succeed either by reuse of common source- or assembly-codelets. The method explicitly represents the performance of phases and whole computations, so solutions can yield extensive architectural insight (Sect. 2.6). The method's speed of solving codesign problems depends on solving LP problems, doing sensitivity analyses, and exploring design space in various ways. Several statistical methods are emerging [7] that may help in reducing the time and enhancing the insights of design space exploration.

Another codesign issue is application performance enhancement. Potential approaches are given in Sects. 2.5.3 and 2.5.6 to find hotspots by roughening profiles across whole computations. Many papers discuss the two- node case [1] seeks memory-processor *balance* (Definition 2.1) through formulas to analyze loops for compiler transformations. Using two node *capacity* Eq. (2.29b) with variable μ_{xy} and constant C_x , switches the Fig. 2.5 labels (Sect. 2.5.3); [10] explores the

$C_x = C_m = B_m$ case (roofline model). For several algorithms, [8] uses memory access intensity analysis—equivalent to *BW sensitivity* (Eq. (2.47)) with saturated memory—to predict when blocking performance-sensitive loops will be beneficial.

The potential to partition design space and move toward specialized systems for distinct applications exists within this method. This can be done manually by iteratively removing similarly performing phases. Perhaps algebraic analysis may lead to semi-automatic methods of partitioning the computational parameter matrix.

2.9 Conclusions

A number of codesign problems have been posed, together with capacity-based methods of finding BWs of HW system nodes that satisfy given goals, for a given set of computations. System recommendation is a related problem, i.e. for a fixed set of computations, select one of several specific systems as the best in *perf/cost*. Also, codesign can be expressed as solving for SW variables in terms of fixed HW. The ideas presented can be used for many specific problems, but there are some underlying commonalities:

1. Top-down codesign of optimal systems
 - Mixed fidelity modeling allows focus on exactly those parts of the HW system of interest
 - All computations are modeled by weighted combinations of SW repository codelets
2. Simultaneous use of comprehensive load and BW information
 - LP equations are optimized faster than discrete event simulation, combining SW and HW specifics
 - Global sweeps of 3D codesign space show parametric relations among many optimal design points
3. Design of robust, focused-system families under uncertainty
 - Perfect solutions ($B^{\text{waste}} = 0$) for single phases, optimal designs for application classes
 - Pre-Si exploration of design sensitivities; market-segment design partitioning.

Key features of the approach include:

- Rich codelet set relative to benchmark/trace-driven simulation helps prevent application performance regressions
- Capturing system-wide interactions avoids the local optimization traps typical in component-wise design
- Automating the process overcomes design complexities that overwhelm human designers
- Meeting infeasible goals with higher-performance synthesis, only when needed.

The *consequences* are savings of human design and CAE machine time, as well as better system designs.

Acknowledgements David Wong and Ahmed Sameh provided many insights in developing this material; David Wong designed and implemented the CAPE tool.

References

1. Carr, S., Kennedy, K.: Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. Program. Lang. Syst.* **16**, 1768–1810 (1994). doi:<http://doi.acm.org/10.1145/197320.197366>
2. Emer, J., Ahuja, P., Borch, E., Klauser, A., Luk, C.K., Manne, S., Mukherjee, S., Patil, H., Wallace, S., Binkert, N., Espasa, R., Juan, T.: Asim: A performance model framework. *Computer* **35**, 68–76 (2002). doi:<http://doi.ieeeecomputersociety.org/10.1109/2.982918>
3. Jalby, W., Wong, D., Kuck, D., Acquaviva, J.T., Beyler, J.C.: Measuring computer performance. In this volume
4. Kuck, D.: Computer system capacity fundamentals. Tech. Rep. Technical Note 851, National Bureau of Standards (1974)
5. Kuck, D.: *The Structure of Computers and Computations*. Wiley, New York (1978)
6. Kuck, D.J., Kumar, B.: A system model for computer performance evaluation. In: *Proc. 1976 ACM SIGMETRICS Conf. on Computer Performance Modeling Measurement and Evaluation*, pp. 187–199. ACM, New York (1976). doi:<http://doi.acm.org/10.1145/800200.806195>
7. Lee, B., Brooks, D.: Spatial sampling and regression strategies. *IEEE MICRO* **27**, 74–93 (2007). doi:<http://doi.ieeeecomputersociety.org/10.1109/MM.2007.61>
8. Liu, L., Li, Z., Sameh, A.: Analyzing memory access intensity in parallel programs on multicore. In: *Proc. 22nd Annual Int'l. Conf. Supercomput., ICS '08*, pp. 359–367. ACM, New York (2008). doi:<http://doi.acm.org/10.1145/1375527.1375579>
9. Uhlig, R.A., Mudge, T.N.: Trace-driven memory simulation: A survey. *ACM Comput. Surv.* **29**, 128–170 (1997)
10. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* **52**, 65–76 (2009). doi:<http://doi.acm.org/10.1145/1498765.1498785>

<http://www.springer.com/978-1-4471-2436-8>

High-Performance Scientific Computing

Algorithms and Applications

Berry, M.W.; Gallivan, K.A.; Gallopoulos, E.; Grama, A.;

Philippe, B.; Saad, Y.; Saied, F. (Eds.)

2012, XIV, 350 p., Hardcover

ISBN: 978-1-4471-2436-8